EEXIST

*A learning system
based on blending
computation
continuity
and karma*

2nd Edition

*including 2-D EEXIST
and a study of introducing
free will into a
derterministic system*

Dr. Nicholas J. Macias
July 2021

Preface

This book is, in many ways, a continuation of work that was begun in the 1980s by Jimmy Hargrove, Larry Henry, Murali Raju and myself. In the midst of conversations about artificial intelligence, data flow machines, microcode and architecture tuning, self-awareness, the halting problem, and a variety of other topics, an idea emerged for a new type of computational building block: a reconfigurable, *self*-configurable platform named the PIG ("Processing Integrated Grid"), later called the Cell Matrix.

17 years later, the first major change to the Cell Matrix was conceived as an analog, continuous-valued version. Several years later, this was more-fully developed as the "Songline Processor."
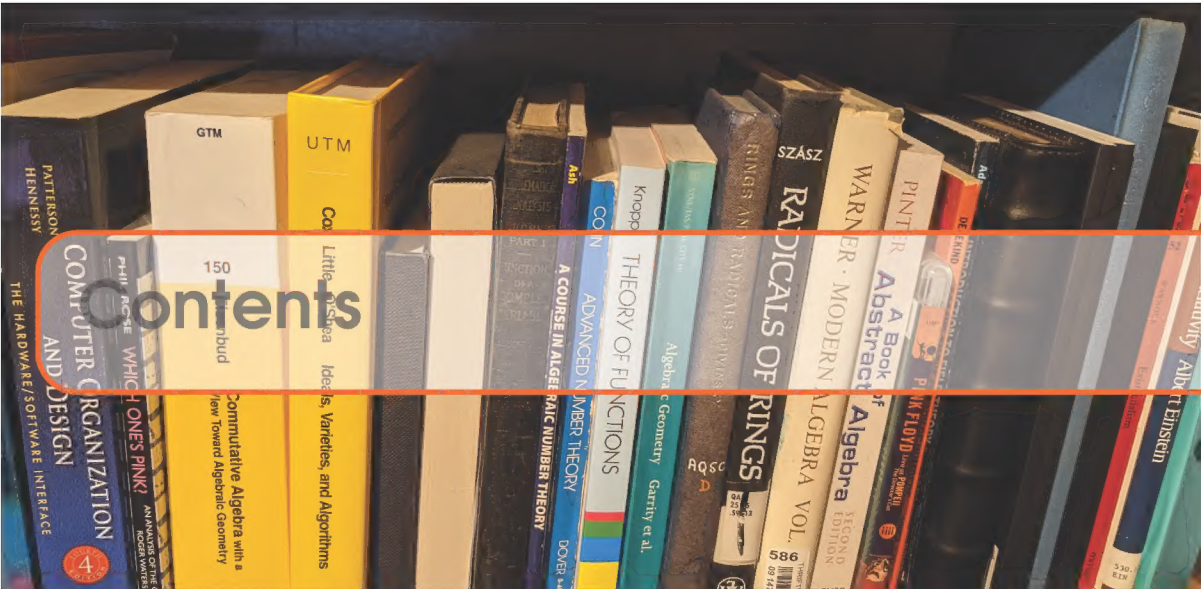
In 2015, EEXIST was developed. This is not a simple extension of prior ideas, but rather a re-imagining of computation from the ground up, influenced by these previous works. The result is a system that barely resembles a computational engine. EEXIST is an abstraction of a highly simplified machine, infused with the central ideas of the Cell Matrix and Songline Processor, modified to incorporate concepts from the real world such as continuity of space and time. The result is a system that is difficult to understand, currently impossible to program in any traditional sense, but seemingly very rich in the behavior it can exhibit.

It is an ongoing challenge to understand the nature of EEXIST, and to figure out how to work with it in its most general, abstract form. This effort is only just beginning, yet already seems to suggest an interesting, useful architecture, applicable to a number of different problem areas. The present work is a description of some aspects of this early work. What is contained herein is not the end of the story, it is only the beginning. My hope is that the interested reader may perhaps find a starting point here for further research in this area.

I am indebted to many friends and colleagues for numerous discussions and brain-storming sessions related to this and similar topics. My students at Clark College have been a particular source of insight and inspiration in this work. Special shout-out to Caleb Hooper, who ported the entire EEXIST system to a 3500+ core GPU, and whose work can be found in Appendix A of this text.

My greatest thanks go to my fiancée, Corinna Dollar, who has been a continual source of joy, hope, love and inspiration in this and all aspects of my life. I love you!

Nicholas Macias
Vancouver, Washington, USA
July 2021

Contents

I   Part One - Theory

Part One - Theory

1. Introduction

This manuscript describes a work in progress. It is not a complete story, nor is it a story whose conclusion has been reached. It is a description of an ongoing research effort that, in many ways, is over 30 years old.

The ideas presented herein may not seem useful at first. The system – EEXIST – is simultaneously difficult to use and control, and limited in demonstrated applications. It does not do any one thing better than existing systems; it is not a drop-in replacement for a von Neumann architecture. Nonetheless, it is a potentially promising direction in which to explore interesting concepts related to extending our notions of computation.

The point of this work is to expose new ideas, new directions in the field of computing. EEXIST feels like a very different approach to computation and control. Moreover, it is not a randomly-conceived architecture, but rather one that has been deliberately designed based on lessons learned and observations of the real world. Successful application of the system to a variety of problems is interesting beyond the solutions to those problems themselves. It is more the fact that such an unusual architecture, which is so difficult to imagine controlling, can in fact perform these algorithms. This is an important point to remember throughout: it is not the solutions themselves that are intriguing, *it is the*

challenge of understanding the system at large that is the main focus of this research.

1.1 Quick View of EEXIST

While carefully reading this manuscript (at least Part One) should give a fairly complete picture of EEXIST, there are doubtless some readers who hope to glean at least a high-level view of the system from this introduction. To that end, here is an attempt at a one-paragraph synopsis of EEXIST.

The idea of this architecture is to define a system where time and space are continuous; where the effect of each action is felt immediately everywhere; and where there is no distinction between subject and object, because everything is at all moments acting on and being acted on by everything in the system.

1.2 Chapter Breakdown

The goal of this text is to expand on the above description, to help the reader understand and explore EEXIST (rather than just apply it to problems). It is thus important to understand the background of the system. Part 1 of this text discusses the theory of EEXIST through a number of topics. Chapter 2 describes a very simple type of computer: a Transfer Machine, also called a Zero-Bit Computer (ZBC), presented as a starting point for the development of EEXIST. Chapter 3 reviews relevant background on the Cell Matrix and the Songline Processor, which are used in Chapter 4 to specify design goals for an extended ZBC. Chapter 5 presents a model for this extended system, based on a system of chemical transfers.

Part 2 covers a series of experiments involving application of EEXIST to different problems, including implementation of digital logic; frequency detection and generation; game playing; and control of creatures in a simulated competitive environment.

Part 3 discusses open questions and next steps.

It should be noted that each of these chapters, rather than being a final word on any of these topics, is a launching-off point for future research. The particulars explored in this manuscript represent one set of possibilities in a much higher-dimensional design space.

1.3 Exercises

Some chapters include exercises at the end of the chapter. These are things to think about or try that may help in understanding the material presented in the chapter.

2. Transfer Machines

What is a computer? Early computers were humans: "computer" was originally a job description. The term "electronic computer" was used to differentiate artificial computing engines from human ones. Computers perform calculations, but more generally they execute *algorithms*. This however is a high-level view of what computers *do*. It does not answer the question "What is a computer?"

At a lower level, computers are systems that store information and process information based on stored instructions. At a machine-code level, this is still a reasonable description. But at a lower level – closer to the hardware – the description changes. Consider a microcoded machine, such as the VAX-11/780 [2]. Inside the CPU, below the level of the machine code, is a *micromachine*. A series of 96-bit microinstructions direct the behavior of the hardware in order to *interpret* the binary machine code. The microinstructions have one main purpose: *to direct the movement of data among the components of the datapath* .

2.1 A Zero-Bit Computer

Note that most of the code in this chapter can be found here [26].

A standard question in a Digital Logic class is: "how many different instructions can be represented with n bits?" The answer (2^n) suggests that more than a few bits is often sufficient, at least in terms of *opcodes*. A similar question (sometimes posed in my Discrete Structures class) is: "what is the fewest number of bits required for coding an opcode in a generally-useful computer architecture?" If we reduce this to the case of a transfer machine, the somewhat surprising answer is 0: the system only requires one instruction ("transfer") and, being the only instruction, requires no bits to code it. Each instruction is understood to be a transfer, and thus only the operands need to be expressed. Hence the term "Zero-Bit Computer" (or "ZBC") is sometimes used to describe such a system.

A ZBC can be viewed as a large memory, containing pairs of addresses, each pair containing a source ("SRC") and a destination ("DST") address. The understanding is that, beginning with a first pair, this memory describes a series of transfer operations that are to take place, copying the contents of the SRC location to the DST location. For reasons that will be made clear below, there is only a single memory, containing both the transfer instructions and the data being transferred. In other words, *the transfer instructions are themselves potentially subject to transfer*.

To do useful work on a ZBC, we require a way to do things such as arithmetic operations. Computational blocks – ALUs, multipliers, etc. – are memory-mapped into the address space, so that, for example:

- transferring to address 1011 might copy a value to the A input of an ALU;
- transferring to address 1012 might copy a value to the B input of an ALU;
- transferring to address 1010 might copy a value (meaning *add*) to the FUNCTION SELECT input of an ALU; and
- transferring from address 1013 might copy the result of the specified function on the two inputs (i.e. their sum).

By having this and similar blocks (for input and output, etc.) mapped into the ZBC's address space, one can write code that performs various procedural steps. For example (using the above-described ALU), listing 2.1 shows code that will add the contents of locations 100 and 101 and store their sum in location 102.

Listing 2.1: ZBC Code to Add Two Variables

```
Address   Contents
   0         100      ; read  the  first  number
   1        1011      ; copy  to  ALU(A)
   2         101      ; read  the  second  number
   3        1012      ; copy  to  ALU(B)
   4        2000      ; use  loc  2000  to  save  a  constant  (1)
   5        1010      ; copy  the  literal  1  to  ALU(FUNC)
   6        1013      ; read  the  ALU's  output
   7         102      ; and  save  in  location  102
```

```
2000              1      ; this  is  a  literal  used  to  specify  ADD
```

This code assumes the ZBC begins executing by reading locations 0 and 1 from memory, and copying from the address specified in location 0 to the address specified in location 1. If we view *MEM* as an array describing the memory's contents, the exact behavior is $MEM[MEM[0]] \rightarrow MEM[MEM[1]]$. In other words, all memory references are indirect: the transfer is not from $MEM[0] \rightarrow MEM[1]$ but rather from $MEM[100] \rightarrow MEM[1001]$. This has the desired effect of copying the first variable to be added (stored in memory location 100) to the ALU's A input.

Following this transfer, the ZBC reads memory locations 2 and 3, and transfers from $MEM[101] \rightarrow MEM[1012]$. This process continues with successive pair of memory locations. After the pair at $MEM[6]$ and $MEM[7]$ are read and processed, the sum of the data in $MEM[100]$ and $MEM[101]$ should be stored in $MEM[102]$. The ZBC has performed an addition of two variables.

Note that if the ZBC executes 1000 consecutive pairs of instructions, it will then attempt to execute the transfer request stored in $MEM[2000]$ and $MEM[2001]$. Since $MEM[2000]$ was being used to store a literal value, executing a transfer based on that stored value would be undesirable. This is, of course, a familiar situation in most stored-program computers with a single memory: *there is no intrinsic differentiation between code and data*. More generally, in Listing 2.1, it's unspecified what should happen after memory locations 6 and 7 are read and processed: whatever happens to be in memory next will be executed. This raises the need for some sort of branch capability.

2.2 Branching in a ZBC

Branching is handled very simply, by **mapping the program counter (PC) itself into the memory of the ZBC**. In the simulated implementation, the PC is mapped to memory address 65,535. Thus, the following modification to listing 2.1 causes the code to loop repeatedly:

Listing 2.2: ZBC Code to Add Two Variables and loop forever

```
Address    Contents
   0          100      ; read  the  first  number
   1         1011      ; copy  to  ALU(A)
   2          101      ; read  the  second  number
   3         1012      ; copy  to  ALU(B)
   4         2000      ; use  loc  2000  to  save  a  constant  (1)
   5         1010      ; copy  the  literal  1  to  ALU(FUNC)
   6         1013      ; read  the  ALU's  output
   7          102      ; and  save  in  location  102
   8         2001      ; read  the  literal  0
   9        65535      ; and  copy  to  the  PC
2000            1      ; this  is  a  literal  used  to  specify  ADD
2001            0      ; this  is  a  literal  used  for  branching
```

The additional instruction (at memory locations 8 and 9) specifies a transfer from $MEM[2001] \rightarrow MEM[65535]$, which copies a 0 to the PC, thus causing the instructions to be re-executed beginning with location 0.

2.3   Conditionals in a ZBC

With procedural statements and looping, we almost have a complete programming language: the last piece we need is a way to do conditionals. This is already available though, since we can manipulate the PC based on the value of a variable. For example, listing 2.3 shows a simple if/then/else statement. If the contents of memory location 100 is 0, the code will jump to location 10; if the contents is 1, the code will jump to location 20.

Listing 2.3: ZBC Code for Conditional Execution

```
Address    Contents
   0          100     ; read the branch variable (must be 0 or 1)
   1         1011     ; copy to ALU(A)
   2         2002     ; read the constant 2004
   3         1012     ; copy to ALU(B)
   4         2000     ; read the constant 1
   5         1010     ; copy to ALU(FUNC)
   6         1013     ; read ALU's output (=either 2004 or 2005)
   7            8     ; and save in next instruction SRC
   8         0000     ; placeholder − copy from 2004 or 2005
   9        65535     ; to the PC. This causes a jump to 10 or 20
 2000            1     ; this is a literal used to specify ADD
 2002         2004     ; another literal
 2004           10     ; branch target if MEM[100]=0
 2005           20     ; branch target if MEM[100]=1
```

Listing 2.3 uses the same ALU operations to add the contents of $MEM[100]$ to the constant 2004, giving a sum of either 2004 or 2005. That sum is saved in MEM[8], so that when that instruction is executed, it specifies either a transfer form $MEM[2004] \rightarrow MEM[65535]$ or $MEM[2005] \rightarrow MEM[65535]$, which copies either 10 or 20 to the PC, causing the next instruction to be executed from either location 10 or 20.

One complication to the above code is that $MEM[100]$ must be exactly 0 or 1 (since it's basically being used as an index into a branch table). Using a comparator (mapped into the ZBC's address space, just like the ALU) will work well here, to generate a 1 or 0 based on a desired comparison.

2.4   A ZBC Programming Language

This is all perhaps a bit awkward, but can nonetheless be used to methodically code conditional statements which, along with branches, allow implementation of most any

algorithm. To write code more easily, a simple programming language can be defined as follows:

- two numbers separated by a colon (:) represent a location/contents pair to store in memory;
- a single number represents a number to be stored in the next successive location in memory;
- everything between a semicolon (;) and the end of the line is a comment;
- blank lines are ignored

So, for example, the code in listing 2.3 could be written as shown in listing 2.4 (comments have been dropped):

Listing 2.4: Compiled ZBC Code for Listing 2.3

```
0:100
1011
2002
1012
2000
1010
1013
8
0
65535
2000:1
2002:2004
2004:10
20
```

2.5   A ZBC Interpreter

Using this shorthand, we can write code suitable for interpretation in a simple ZBC interpreter. Listing 2.5 shows a short C program that ingests code such as 2.4 and interprets its execution. You can find this in the file "xm.c" available here [26]:

Listing 2.5: Code for ZBC Interpreter

```c
#include <stdio.h>

int mem[65536]; // main mem!
#define PC mem[65535] // PC is actually stored at end of mem
#define DEBUG (mem[65534]) // set to 1 to turn on debugging

main(int argc, char **argv)
{
  FILE *fp;
  char buffer[120];
```

```
    int  addr=0,data,t1,t2,from,to,status;

// load program into memory
  if (argc==2) fp=fopen(argv[1],"r"); else fp=stdin;
  while (NULL != fgets(buffer,120,fp)){
    if (2==(status=sscanf(buffer,"%d:%d",&t1,&t2))){  // addr:data
      addr=t1;data=t2;
    } else data=t1; // just data
    if (status >=1) mem[addr++]=data;
  }

// main processing loop...
  while (1){
    if (DEBUG)  printf("%d:%d[%d]=>%d\n",PC,mem[PC],
                        memRead(mem[PC]),mem[PC+1]);
    from=mem[PC++];PC&0xffff;
    to=mem[PC++];PC=PC&0xffff;
    memWrite(to,memRead(from));
  }
}

// memory-mapped-hardware simulation

int alufunc=0;  // 1=+,2=-,3=*,4=/
int aluA=0,aluB=0;
int compfunc=0;  //1=> 2==
int compA=0,compB=0;

memWrite(int loc, int data)
{
  switch (loc){
// 1000=DISPLAY
    case 1000:display(data);break;

// 1010=ALU
    case 1010:alufunc=data;break;
    case 1011:aluA=data;break;
    case 1012:aluB=data;break;

//1020=COMPARATOR
    case 1020:compfunc=data;break;
    case 1021:compA=data;break;
    case 1022:compB=data;break;

// If not HW, then just write to memory
    default:mem[loc]=data;
```

```
  }
}

memRead(int loc)
{
  switch(loc){
// ALU
    case 1013: // ALU
      return(aluvalue());

// COMPARATOR
    case 1023: // comparator
      return(compvalue());

// Not HW: just read from memory
    default:return(mem[loc]);
  }
}

// support code

display(int data)
{
  printf("%d ",data);fflush(stdout);
}

aluvalue() // calculate ALU value
{
  switch(alufunc){
    case 1:return(aluA+aluB);
    case 2:return(aluA-aluB);
    case 3:return(aluA*aluB);
    case 4:if (aluB != 0) return(aluA/aluB);
           return(0);
  }
  return(0);
}

compvalue() // calculate comparator value
{
  switch(compfunc){
    case 1:return((compA>compB)?1:0);
    case 2:return((compA==compB)?1:0);
  }
  return(0);
}
```

The main processing loop of this code is only 3 lines, reflecting the very simple nature of the ZBC's architecture. The code also includes some memory mapped hardware, including:

- a display at $MEM[1000]$;
- an ALU at $MEM[1010 - 1013]$;
- a comparator at $MEM[1020] - MEM[1023]$;
- the PC at $MEM[65535]$; and
- a debug flag at $MEM[65534]$.

2.6   Sample ZBC Programs

Given this very simple architecture, programs for performing even modest tasks can be fairly lengthy. Listing 2.6 shows code for counting down from 10 to 0, displaying each value on an output port (the display mapped to ZBC address 1000).

Listing 2.6: ZBC Code to Count Down From 10 to 0

```
65535:0  ;  Set  PC=0
65534:0

;  constants
511:1
512:10
513:0
514:2
515:150  ;  for  infinite  loop
520:110
500:0      ;  X

;  start  of  program
0:512
500        ;  x=10

;  LOC  2
500
1000       ;  Display  X
514        ;  constant=2
1010       ;  ALU  func=sub
500
1011
511
1012
;  now  1013=X-1
1013
500        ;  X=X-1
```

```
;  see  if  X=0
514
1020      ;  comparator  "="  function
500
1021      ;  A
513       ;  0
1022      ;  1023  shows  X==0

511
1010      ;  ALU  "+"
520
1011      ;  110
1023      ;  0  or  1
1012      ;  to  B...[1013]=110(x<>0)  or  111  (X==0)

;  branch  to  loc  100
99
65535
99:100

;  here  is  loc  100
100:1013
101:102
102:000
103:65535  ;  if  X<>0  this  branches  to  [110]  else  [111]

110:120  ;  branch  to  120  if  x<>0
111:150  ;  branch  to  150  if  x==0

120:514
65535      ;  LOOP!

150:515
65535      ;  stay  here
```

A more complex example is shown in listing 2.7, which shows code for generating prime numbers.

Listing 2.7: ZBC Code to Generate Prime Numbers

```
65535:0  ;  initial  PC=0
65534:0  /  disable  debugging  :)

;***  HW  addresses
;1000:  DISPLAY
```

```
;1010: ALU func  (1:C=A+B;2:C=A–B;3:C=A*B;4:C=A/B)
;1011: A
;1012: B
;1013: C

;1020 Comparator relation R (1:>,2:=)
;1021: A
;1022: B
;1023: A r B

500:3   ; A
501:1   ; B
502:0   ; X

; Constants
511:1
512:2
513:3
514:4

; Program starts at 0

0:511
1:501   ; b=1

; B=B+2
2:511
1010 ; ALU set to +
501
1011
512
1012
1013
501 ; B=B+2

; X=A/B
514
1010 ; ALU set to /
500
1011
501
1012
1013
502 ; X=A/B

; is B>X?
```

```
511
1020 ; comparator set to >
501
1021
502
1022
; 1023 is 1 if b>x else 0
; use to to determine a branch address

511
1010 ; ALU=+
1023
1011
100 ; =101
1012
; 1013 (adder output) is either 102 (B>X) or 101

; jump to 110 (just so we know our PC!)
109
65535 ; read from loc 109 (=110) to PC
109:110 ; loc 109 contains 110

100:101
101: 150 // B <= X code goes at 150
102: 120 // B>X code goes at 120

; (loc 110: continue here)
110: 1013
111: 112 ; loc 112 contains either 101 or 102
112: 0 ; contains 102 if B>X
65535  ; now the PC is 120 if B>X

; PRIME!
120: 500
1000 ; display the prime number
511
1010
512
1011
500
1012
1013
500 ; A=A+2

511
501 ; B=1 (again)
```

```
510 ; constant 0
65535 ; Jump to loc 0


; See if X==int(X)
150:513
1010 ; ALU func=*
502
1011 ; X
501
1012 ; B
; [1013]=B*X
512
1020 ; COMP r is "="
1013
1021
500
1022
; [1023]=1 for composite, 0 for continue
179
65535 ; branch to [179] (=180)

; setup for conditional branch to 200 (1023=0) or 220 (1023=1)
165:167 ; constant
166:0 ; either 167(cont) or 168 (comp)
167: 200 ; (continue)
168: 220 ; (comp)

179:180
; branched here...
1023 ; 0 or 1 (continue or composite)
1011
165
1012
511
1010 ; [1013]=167 or 168
1013
188
; here is loc 188
0 ; filled in with either 167 or 168
65535 ; branch to that loc

; continue
200:512 ; branch to loc 2 (B=B+2)
65535
```

```
; composite
220:511
1010
500
1011
512
1012
1013
500 ; A=A+2
510
65535    ; branch to loc 0 (B=1)
```

As with many things at this level, this code is easier (though not easy) to write than to read! While the ZBC is not very practical for general purpose programming, it represents a pared down, minimalist architecture that serves as an ideal starting point from which to re-build our notion of computation. Before we can do this, we must review some of the ideas that drove this redesign: specifically, the Cell Matrix and the Songline Processor (SLP)[3]. These are the topic of the next chapter.

2.7 Exercises

Exercise 2.1 Download the ZBC code (xm.c), compile it, and run the count and prime test files (all of these are available here [26].
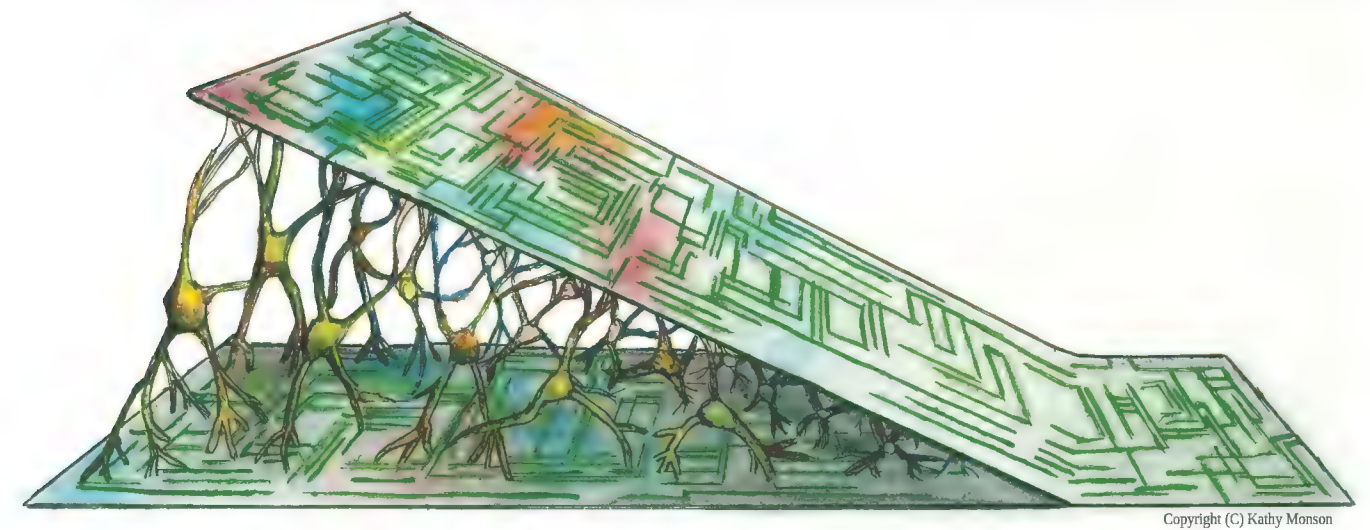
Exercise 2.2 Write a ZBC program for adding two numbers.

Exercise 2.3 Write a ZBC program for printing the larger of 2 numbers.

Exercise 2.4 Write a ZBC program for adding a set of numbers.

Exercise 2.5 What is a minimal set of memory-mapped hardware that allows a ZBC to be used for general-purpose computing?

3. Cell Matrix/SLP Background

Extensive details on the Cell Matrix and the Songline Processor can be found at the main research website http://songlinesystems.com [1]. What is presented here is a high-level summary, specifically of those features that lead to the development of EEXIST.

3.1 Cell Matrix

The idea of reconfigurable logic is straightforward. Software is changeable, morphable, able to be modified with its behavior changing accordingly (that's the "soft" aspect of it). Hardware, in contrast, is rigid, fixed in form and function, and generally difficult to modify without some sort of invasive procedure (de-soldering, re-wiring, etc.) (that's the "hard" aspect of it). Reconfigurable hardware combines the best of these two models, offering the speed of a hardware system with the flexibility of a software system. Thus devices such as field programmable gate arrays [14] ("FPGAs") became popular in the 1980s.

While FPGAs extend the notion of software to the hardware domain, early devices lacked one important aspect of software: the ability of software to examine and modify itself. This aspect – which is a hallmark of the stored program computer [15] – offers many advantages over a system whose configuration is controlled only from outside the system.

The Cell Matrix [16] addresses this by endowing the basic reconfigurable elements of the system with the ability to directly read and write the configuration of other elements. Before discussing this ability for *self modification*, it will be useful to discuss the overall structure of the Cell Matrix, as well as its basic configurability.

3.1.1 Cell Matrix Architecture

The Cell Matrix is comprised of a large grid of simple, identical elements called "cells." Each cell has a set of inputs and outputs, connecting it to a fixed set of *neighbors* . The inputs to a cell are processed by the cell's internal program, and the cell generates outputs accordingly. Figure 3.1 shows a set of connected cells.

The program inside each cell is a simple truth table , which combinatorially maps inputs to outputs. In the 2-D example shown in figure 3.1, each cell has 4 neighbors, and thus continually receives a total of 4 bits of inputs from its neighbors. The cell also produces a single output bit to each neighbor, thus requiring 4 output bits. This mapping can be defined by a truth table, as shown in figure 3.2.

Note that this truth table can be defined by the 64 bits in the output columns. These are stored in a per-cell memory and define the basic input-to-output mapping of a cell. Note that *the mapping from input to output is unclocked* : as soon as inputs change, the outputs change in response as soon as possible. Nothing here is synchronized to any sort of global clock. While this may make the design process more complicated than a synchronous one, it offers advantages in speed and flexibility (and synchronicity can also be added via additional mechanisms defined below).

3.1.2 Sample Cell Matrix Circuits

Designing circuits with these cells is fundamentally no different from designing with standard digital logic blocks. One can, for example, configure cells to act as logic gates (AND, OR, etc.). Cells can also be configured to act as simple wires. By combining these in the right way, general digital circuits can be constructed. Figures 3.3 - 3.5 show some sample circuits, along with the truth tables used to configure each cell.

In figure 3.3, a single cell is being used as a one-bit full adder : the truth table is simply set up to produce the proper outgoing sum and carry bits in response to the inputs. Note that the cell has unused inputs and outputs.

Figure 3.4 shows a set of 8 such one-bit adders, situated side-by-side so one adder's outgoing carry is fed to the next adder's incoming carry, thus creating an 8-bit ripple-carry adder]. The inputs are fed in parallel to the north and south, and the parallel sum appears to the south.

Note that this layout could, in theory, be extended to any number of cells/adders/bits:
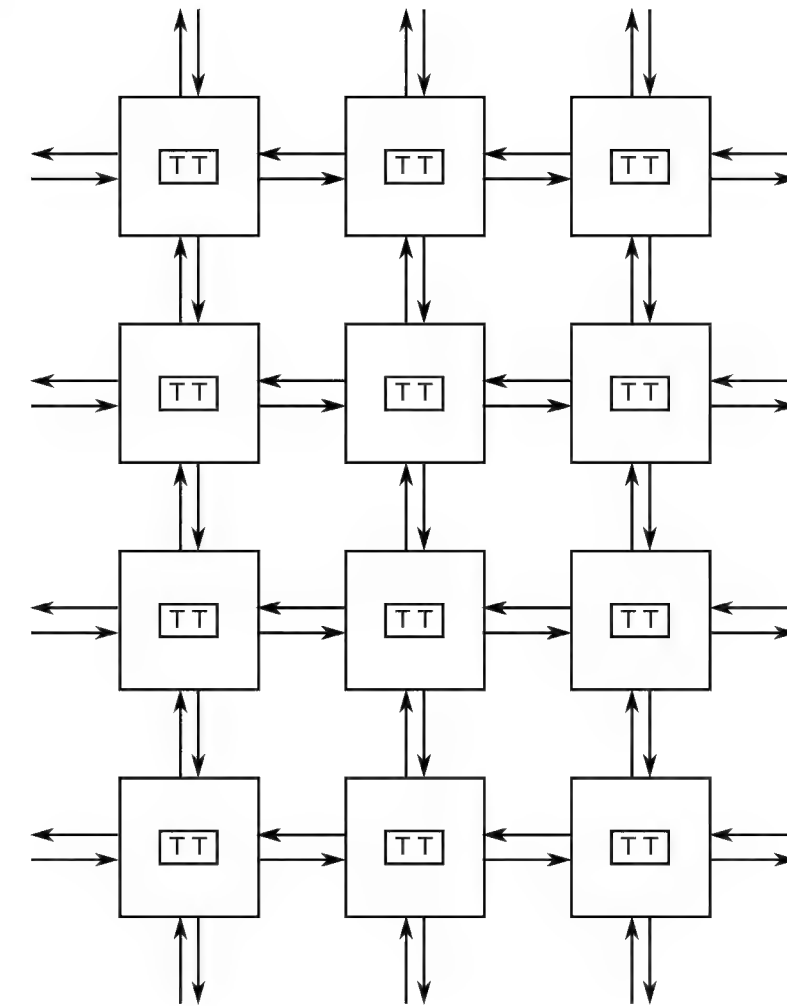
Figure 3.1: Two-Dimensional Matrix of Cells. All blocks are identical except for the program stored inside each one (in the block labeled "TT"). Connections to neighbors are shown with arrows.

1024 cells would produce a 1024-bit adder. While the potential propagation delay from a 1024-bit ripple carry adder is likely to be prohibitively large, the principle is a general one: by carefully designing blocks of cells to be modular, larger circuits can sometimes be constructed simply by placing these clocks together in the matrix. This is one key to *autonomous circuit synthesis*.

Figure 3.5 shows a different type of Cell Matrix circuit, this one comprised of two cells. This configuration implements a simple D flip flop : a one-bit storage element. The setup is straightforward: the cell on the left either sends an incoming bit from the west to the east, or echoes an incoming bit from the east back to the east. The cell on the right provides feedback, reflecting whatever is sent from the cell on the left. When the gate is 1, incoming data is sent to the cell on the right; when the gate drops to 0, that bit becomes

INPUTS				OUTPUTS			
N	S	W	E	N	S	W	E
0	0	0	0	D_{03}	D_{02}	D_{01}	D_{00}
0	0	0	1	D_{07}	D_{06}	D_{05}	D_{04}
0	0	1	0	D_{11}	D_{10}	D_{09}	D_{08}
				...			
1	1	1	1	D_{63}	D_{62}	D_{61}	D_{60}

Figure 3.2: Truth Table For a Cell Connected to Four Neighbors. Neighbors are referenced by their compass directions ("N," "S," "W" or "E") relative to the cell. D_n refers to the nth bit of the truth table.

trapped inside the cells, being passed from one to the other continually.

Thus it is possible to design sequential circuits that operate with a clock an synchronize operations using e.g. standard state machine design techniques. Figure 3.6 shows a more-complex circuit employing logic blocks, flip flops and wires (wires are implemented exactly like other functions, i.e. the equation $D_E \rightarrow D_W$ copies data from the eastern D input to the western D output). This circuit implements a small RAM. One of 16 rows can be addressed with the inputs A_3, A_2, A_1 and A_0. When the WRITE input is high, the 4 data bits $D_3 - D_0$ (from the top of the circuit) are loaded into the selected row of flip flops. When the READ input is high, the selected row's flip flops supply outputs to $D_3 - D_0$ and the bottom of the circuit.

3.1.3　Configuration of Cells: C-Mode

The above description of cells covers their behavior in terms of *data processing*, i.e., transforming inputs to outputs. While this is sufficient for implementing standard digital circuits, it doesn't allow the introspection described above: in particular, it does not explain *how cells are configured*. To allow cells to be configured by other cells, we add an additional input and output to each side of the cell, giving a cell as shown in figure 3.7.

In this cell, the C inputs are used to control the configuration of the cell, as follows:
- if $C = 0$ (called "D-mode"), then the D inputs are processed as described above: they reference a row of the cell's truth table, which contains the outputs that are sent to neighboring cells;
- if $C = 1$ (called "C-mode"), then the corresponding D input is used to supply truth table bits, i.e., to populate the cell's truth table. The corresponding D output is used to read the truth table's current contents.

C-mode operations are clocked via a system-wide clock (which is only used in C-mode, but can be tapped into and utilized by D-mode circuitry).

Figure 3.8 shows the interaction of C- and D-modes, as well as a typical read/modify/write operation. When the cell (configured as a simple wire $DE \rightarrow DW$) is in D-mode,

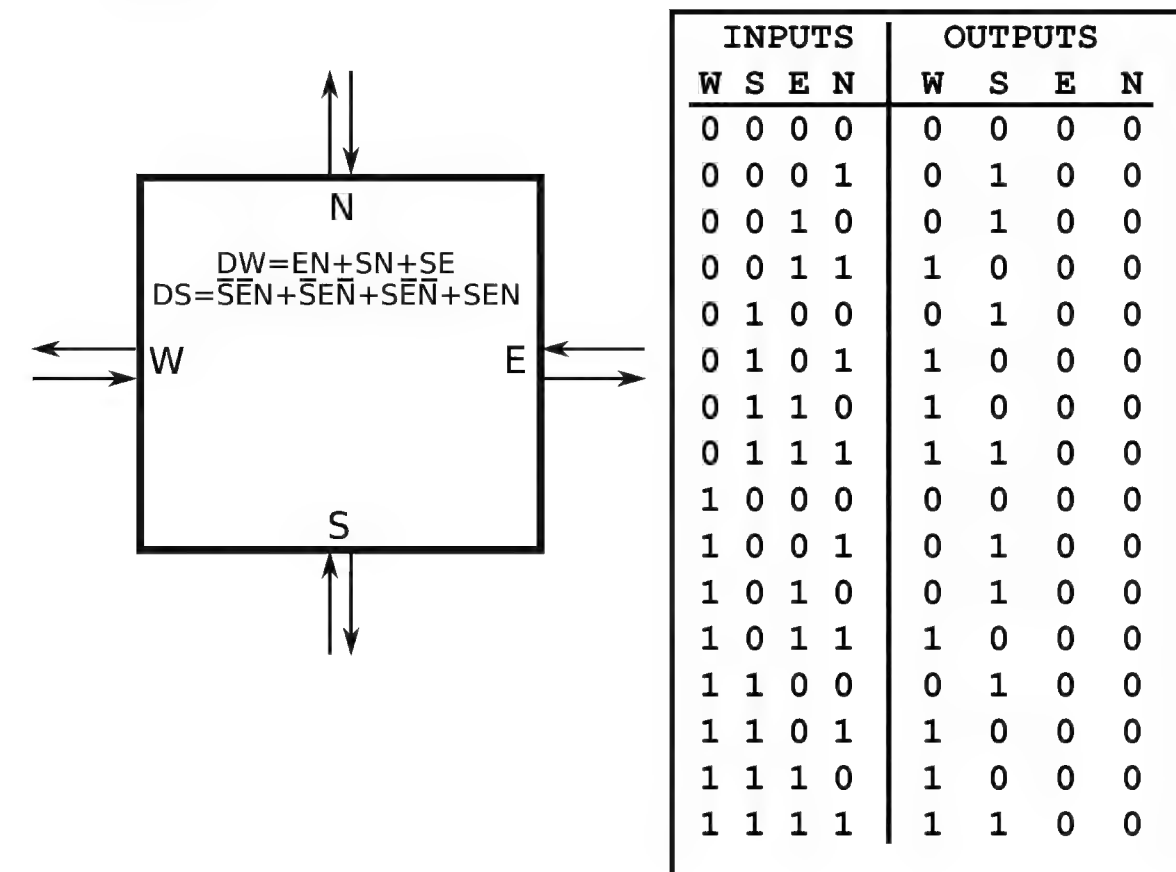| INPUTS				OUTPUTS			
W	S	E	N	W	S	E	N
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	0
0	0	1	1	1	0	0	0
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	0
0	1	1	1	1	1	0	0
1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0
1	0	1	0	0	1	0	0
1	0	1	1	1	0	0	0
1	1	0	0	0	1	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	1	0	0

Figure 3.3: A Single Cell Setup As a one-Bit Full Adder. Incoming bits are supplied to the north and south; the incoming carry is applied to the east; the sum is presented to the south; and the outgoing carry appears on the west. The Boolean equations for the cell's truth table are shown inside the cell.
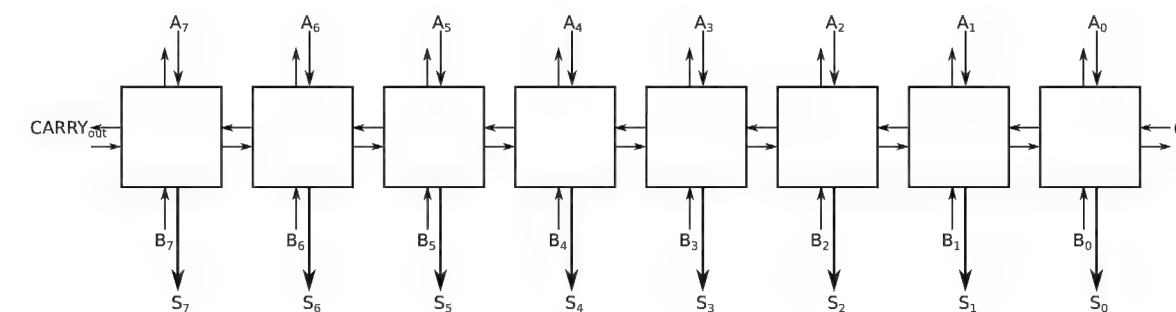

Figure 3.4: 8 Cells Setup As an 8-Bit Adder. Inputs A and B are supplied to the north and south, and the sum S is presented to the south. Each cell is identical to the one shown in figure 3.3.

its eastern output reflects its western input, regardless of the state of the system clock.

GATE

$$DE = WN + E\overline{N}$$

N

D → W   E

S

$$DW = W$$
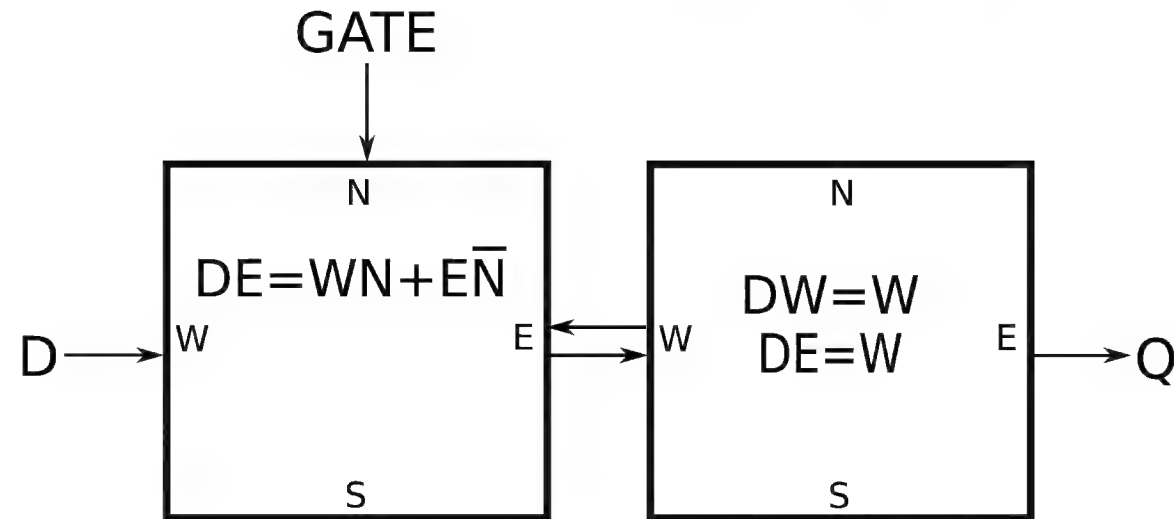$$DE = W$$

N

W   E → Q

S

Figure 3.5: A Data Flip Flop. The D input comes from the west, the clock is presented on the north; Q comes out from the east.

When the cell enters C-mode, the D output changes to reflect the "first" bit (D_{000} according to the pre-defined ordering shown in figure 3.7) in the cell's truth table. On the rising edge of the clock, the D input is sampled and saved inside the cell. On the falling edge, that saved value replaces the first bit in the truth table, and the D output now reflects the second truth table bit. When the C input returns to 0, the cell re-enters D-mode, and the D output now reflects the results of applying the cell's D inputs to its new truth table.

This simple setup allows a number of interesting circuits to be implemented. For example, figure 3.9 shows a *cell reader* . The cell on the right places the target cell into C mode by asserting a 1 to its CW output (which is the target cell's CE input). The target cell sends its current truth table bits out its DE output, which the cell reader ingests from its DW input. The cell reader copies those old truth table bits back to its DW output, which loads them back into the target cell's truth table (thus implementing a non-destructive read). The target cell also copies those truth table bits to its own DS output, where another cell could pick them up and process them.

Figure 3.10 shows a *cell replicator* , which is similar to the cell reader, but with one small change: the CS output is set to 1. With this change, the bits being read from the target cell's truth table will be copied into the cell to the south, thus making that cell an exact copy of the target cell.

Figure 3.11 shows a further embellishment to the cell replicator. In this circuit, the cell replicator is connected to a horizontal row of 3 circuits which copy bits from the west to the south and east, while asserting their CS output. The result is that four copies of the target cell are created (in the second row of the circuit). Note that these copies are created in parallel: the writing of all 4 truth tables occur at the same time. This is thus a *parallel cell replicator* .

It should be noted that the 3 cells on the right of the top row are identical to each other.
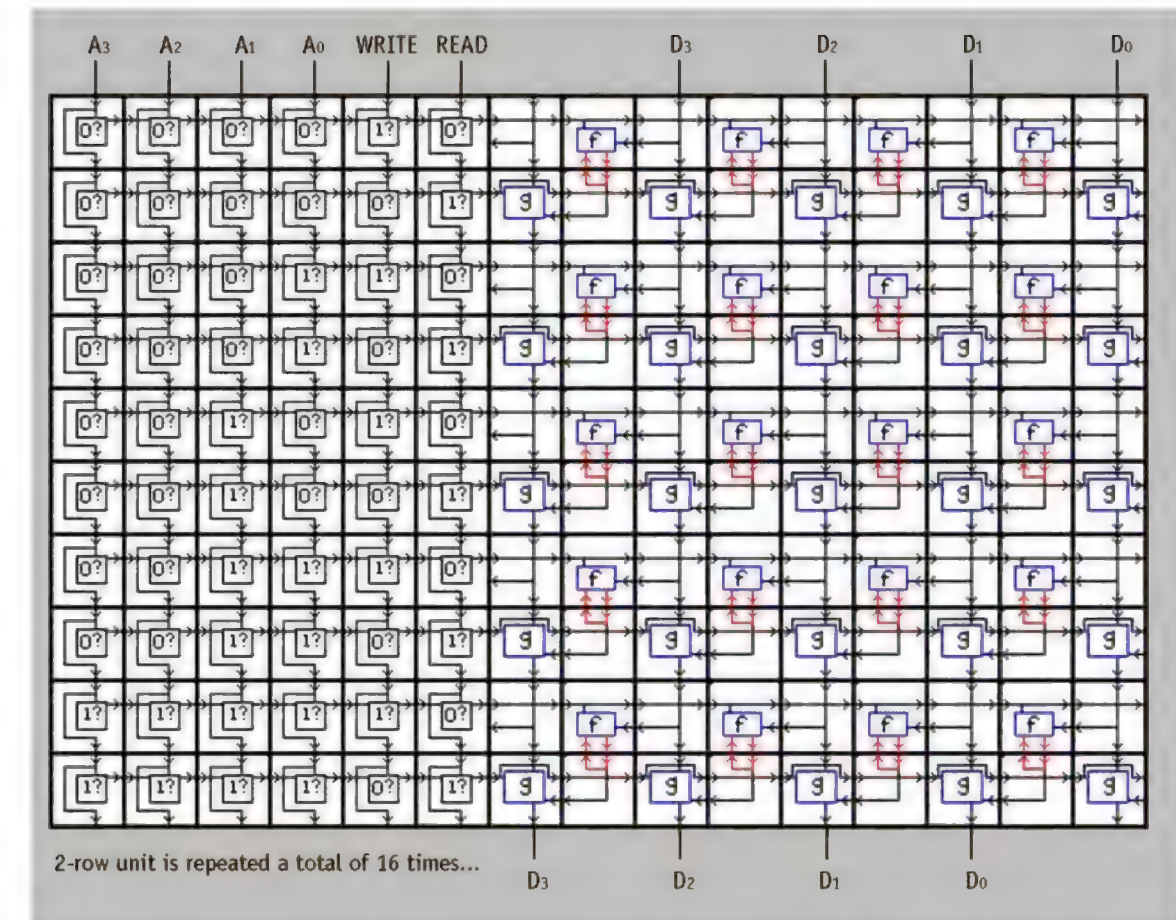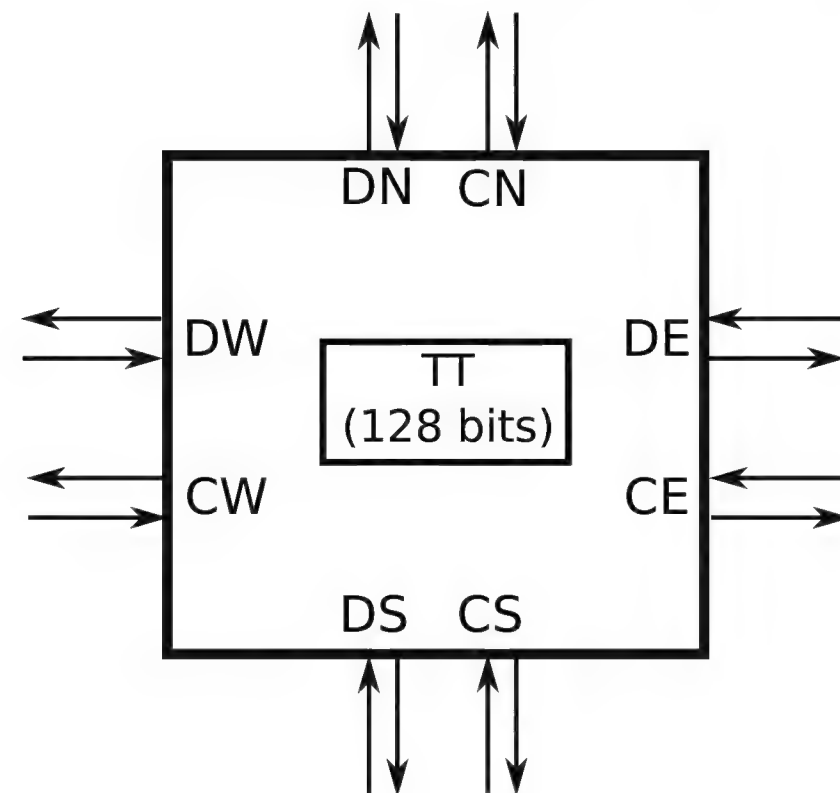
Figure 3.6: More-Complex Cell Matrix Circuit. This layout implements a small memory. The left side of the circuit routes address inputs ($A_3 - A_0$) from top to bottom; the 4th column outputs a 1 to the right when the address matches. That match output is combined with the READ and WRITE inputs to drive the array of flip flips (on the right side of the circuit). Each flip flop is comprised of a cell ("f") and a feedback circuit to its right (similar to the circuit in figure 3.5. Inputs are loaded in response to a match from the address block combined with a WRITE signal. Flip flop outputs are sent to the "g" blocks, which either pass the prior block's output from north to south (if there is no match); or pass the flip flop's output to the south (if the flip flop is being addressed by the $A_3 - A_0$ inputs).

This means that these cells could themselves have been configured in parallel. Of course, that would require *another* circuit to perform *that* parallel replication, so it would seem parallel replication of *n* cells always requires (at least) *n* operations to set it up. In fact, this is not true: done properly, it takes on the order of *n* operations to configure n^2 cells in a 2D matrix. On a 3D matrix, *n* operations are sufficient for configuring n^3 cells. Circuits for doing these better-than-parallel builds are called *"Medusa Circuits"* [17].

Figure 3.7: *Full Cell Matrix Cell. There are* two *inputs and outputs ("C" and "D") connecting this cell to each neighbor. The D lines are used for regular data processing, while the C lines are used for configuring the cell.*

3.1.4 Applications of C-mode

Key features of C and D mode in the Cell Matrix include:

- the process of configuring cells is intrinsic to the overall architecture;
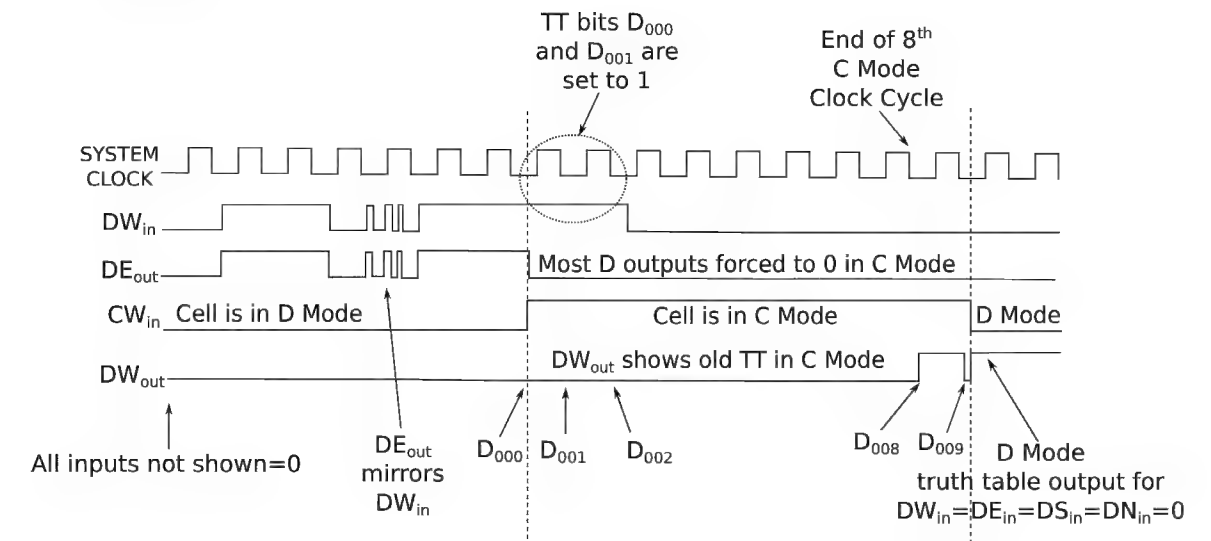- control over cells can be realized *from within the system itself*;

Figure 3.8: *Interactions of C-mode, D-mode and the System Clock. In D-mode, the system clock has no effect on the cell. In C-mode, reading and writing of the cell's truth table is synchronized to the system clock.*
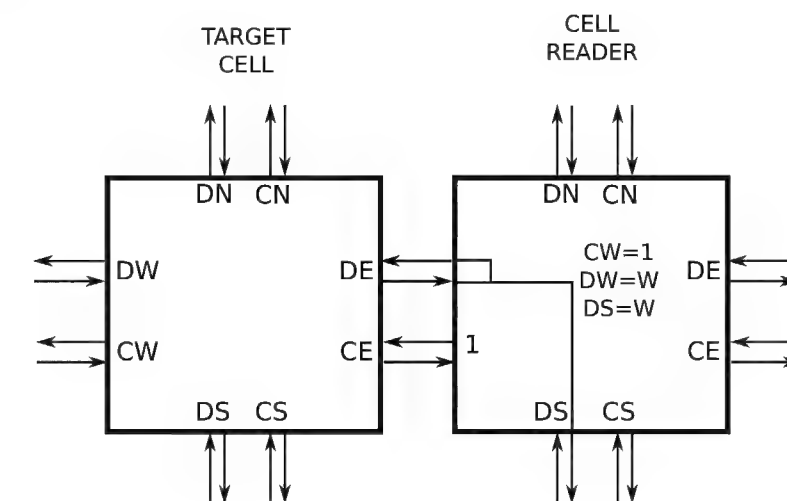

Figure 3.9: *A Cell Reader. The cell on the right reads the truth table from the cell on the left. It feeds this data back to the cell on the left, while also transmitting it to the south.*

- the control is distributed throughout the system;
- controlled and controlling entities are interchangeable ("non-dualism"); and
- the homogeneity of the cellular organization makes the architecture highly scalable .

By properly utilizing these features, a variety of behaviors can be realized in circuits built on the Cell Matrix. The following is a partial list of some of the areas where the Cell Matrix is well-suited:
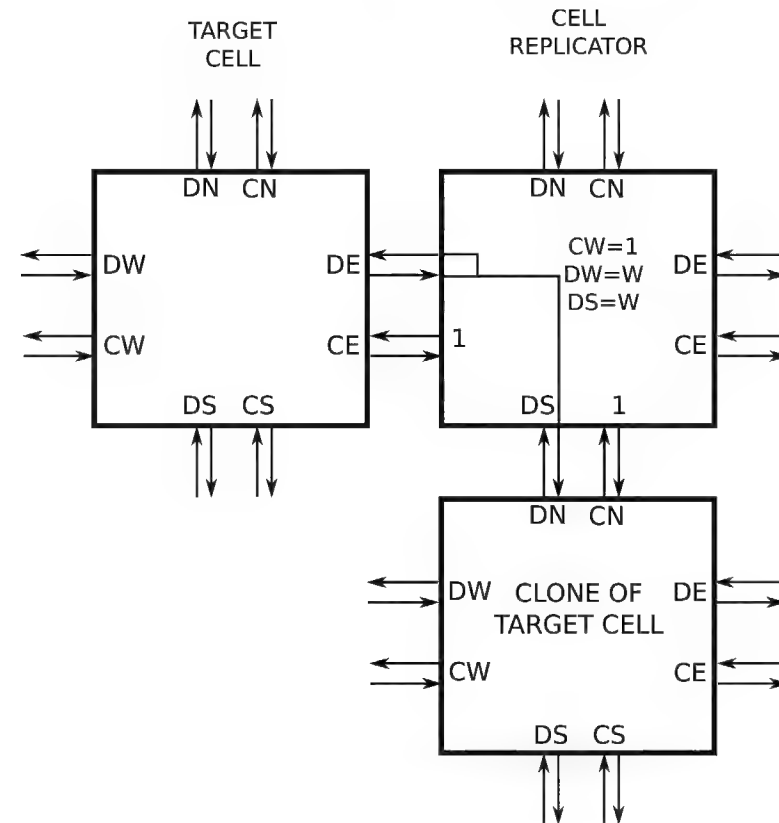
Figure 3.10: A Cell Replicator. The cell replicator reads the target cell's truth table bits, but copies those bits into the truth table of the cell to the south, thus making it an exact copy of the target cell.

- Highly-parallel processing . If a problem can be divided into a set of identical, independent sub-tasks, then groups of cells can be configured to perform one set of sub-tasks, and then replicated to perform multiple sets in parallel. While overhead such as control and communication generally obviate such potential performance gains, the ability to build custom control and communication channels (in parallel) makes it possible to realize significant speedups. This is especially true for so-called "embarrassingly parallel" problems. Examples include massive search problems; simulation of 2- and 3-D systems; finite element analysis; and so on.
- Fault tolerant computing. Much work has been done on using the Cell Matrix's self-configurability to build and run test circuits in order to check for hardware defects. The trick – being able to handle potential errors in the test circuits themselves – plys the self-configurability and non-dualism of the system. Moreover, parallel testing is possible, as is parallel synthesis of the test circuits themselves: see this final report [18] on a NASA SBIR investigating use of the Cell Matrix for autonomous fault handling.
- Adaptable Computing . There are domains where being able to change the micro-architecture of a system at run-time may be useful, for example:
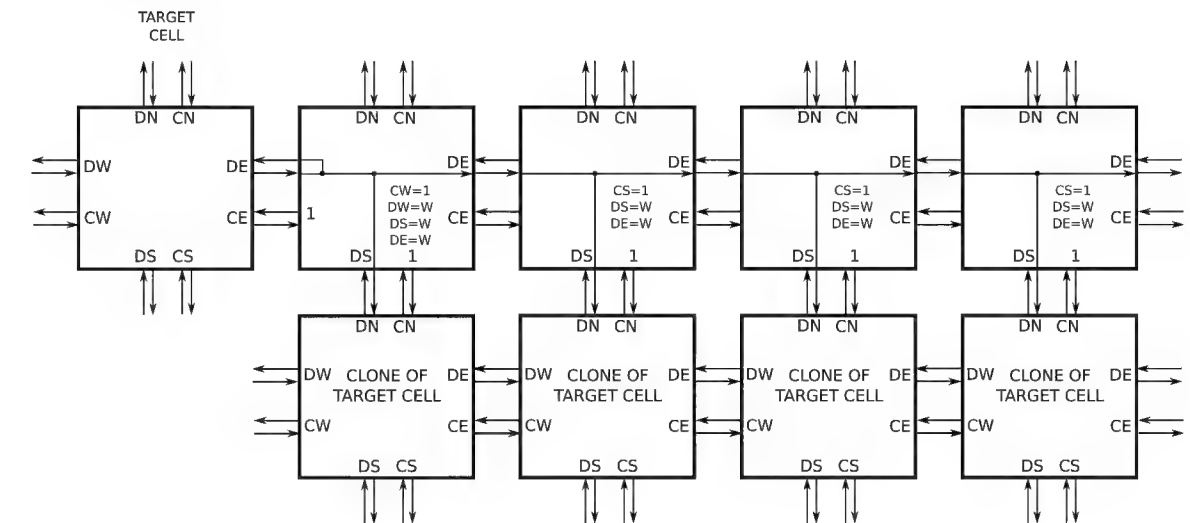
Figure 3.11: A Parallel Cell Replication Circuit. The top row of cells copies truth table bits from the target cell to the four cells on the second row. The replication occurs in parallel: all 4 truth tables are written simultaneously.

- if the same operation is being performed on the elements of a large dataset, multiple hardware instances can be created to operate on the data in parallel;
- if an array processor needs to switch between integer, floating point and character processing, the underlying hardware can be re-purposed to the required datatype;
- if a set of instructions are repeatedly executed, more frequently than other sections of code, they can be cast into hardware, allowing for faster execution (just-in-time hardware synthesis).

These are but a few examples of where adaption of the hardware may be useful. In all cases though, it is the ability of the hardware to *manage the modification of itself* that makes the Cell Matrix a good fit.

- More generally, the interplay between C and D mode allows for the possibility of, say, compiling code into a mix of software and hardware. This is no different from a compiler that takes advantage of a hardware floating point accelerator (FPA): it simply adds a layer of designing and implementing the FPA alongside the machine code.
- Simulating of a Cell Matrix. Yes, this is a thing! While simulation in software is straightforward, it is also inherently slow, as performance degrades in proportion to the number of cells being simulated. With a hardware-based simulation, there is no inherent slowdown as more cells are added to the simulation. But being able to simulate a matrix allows, for example:
  - the ability to freeze the system, or to step it forward slowly while observing its state;
  - the ability to examine internal cells (whose inputs an outputs are normally only

available to neighboring cells);

– the ability to address individual cells and interact directly with their truth tables;

– the ability to reset the state of part of the system without performing a system-wide reset;

and so on.

- The Cell Matrix can be used as a *process improvement driver* for aggressive manufacturing. By targeting the matrix and using its ability for self-analysis, it can help diagnose manufacturing errors in itself, thus helping to debug the manufacturing process itself. One recent example of this is using self-assembly to build a physical matrix from a collection of 3D cells. The cells are manufactured in parallel, and then allowed to self-assemble into a 3D array. In so doing, their orientation is uncontrolled, meaning the final matrix has cells whose orientation is effectively random. This makes it impossible to load truth tables in a meaningful way: building a wire to copy data from a given side to another requires knowing whether those sides are on the north, south, etc. of the cell, and without a fixed physical orientation, this is unknown. However, the cells' ability to introspect allows for circuits and algorithms that let the cells discover and correct-for their own orientation (in parallel). See http://journal.frontiersin.org/article/10.3389/frobt.2016.00002/full [19] for details.

- The Cell Matrix can be embedded alongside other hardware (e.g. MEMS-based systems), and used as a distributed control and communication network, *with all the inherent advantages of massive parallelism, fault tolerance and scalability*.

3.2   Songline Processor

The central themes of the Cell Matrix – self-configuration, homogeneity of the cells, lack of centralized control, nearest-neighbor interconnect – have proven interesting and useful in different ways. Nonetheless, the underlying architecture still derives from more standard models of computation: the system is binary in nature; it employs a clock for reading and writing cell memories; and, despite the interchangeability of modes, each cell is, at any given moment, *either* in D or C mode, i.e., executing its program or being programmed.

The Songline Processor (SLP) is an attempt to retain those unique features of the Cell Matrix, while freeing the architecture from the more standard characteristics. This changes the C and D mode behaviors of the Cell Matrix as follows:

- The values being passed from cell to cell are real-valued vs. binary. This in itself is not such a big difference. In a typical digital system, the binary values are actually voltages, but the circuitry simply restricts those voltages to one of two ranges ("high" and "low"; or 1 and 0). To pass a real-valued signal, just think perhaps of a voltage that can vary anywhere between 0V and 5V. This the D inputs and outputs are real-valued.

- The *C* inputs and outputs are also real valued. This requires a reinterpretation of how a cell's mode affects its behavior. Instead of mode being a binary state (say 1=C Mode and 0=D Mode), the mode is now a mix of C and D modes. In practice,

this means a cell's truth table may be partially perturbed by incoming D values; and D outputs may be a mix of the cell program's output and the program itself.

Implementing these changes requires changing the nature of a cell's truth table. In a cell with binary inputs and outputs, it is possible to exhaustively describe all possible input combinations with a finite number of rows. Adding real-valued outputs doesn't in itself change this: the entries in the truth table's output columns are simply real numbers instead of bits. But since one cell's outputs are connected to another cell's inputs, the *inputs* to a truth table are now also real-valued. This means the set of possible input combinations is (uncountably) infinite. One cannot write a table with a finite collection of rows to represent all possible input conditions: even for a single input, the set of possible input values has the same cardinality as the set of real numbers.

Viewed differently though, this is really not so mysterious: the "truth table" is now simply a real-valued function of real variables (e.g. $z = sin(x^2 + y^2)$). Such a function, however, is not easily stored in a table (unless the input space is discretized). Moreover, even if the function were somehow stored (say, as a curve or a surface), it's not clear how to perform the C mode operations of reading and writing this function. For a curve (e.g. a function of a single variable), one may sweep the input from its minimum value to its maximum value, and thus read/write the entire essence of the function in a fixed amount of time. For 2 or more inputs, there's no obvious (continuous) way to sweep the entire space of input combinations in a finite amount of time. Some sort of space-filling curve may work, but this is currently an unsolved problem.

By abandoning the system clock though, there are other ways to transfer these functions: for example a 2D surface can be used to transform a second surface so as to mimic the first (imagine a 2D sheet of plastic, shaped to represent the Z value of $z = f(x, y)$, and used as a mold to shape a second target piece of plastic).

There are other possibilities as far as implementation: further details are available here [3]. Despite these issues, simulation is still feasible, and has been used to develop and test some sample applications of the Songline Processor. For example, a single cell can be used to multiply two inputs: this is effectively an amplifier. Differentiation, integration, sample-and-hold, amplitude modulation, and other types of signal processing are each easily achieved with just a few cells. This suggests that the Songline Processor is, in some ways, fundamentally different from a traditional digital/von Neumann machine. These differences suggest ways to extend the Zero-Bit Computer, as discussed in the next chapter.

3.3   Exercises

Exercise 3.1 Write the Boolean equations for a single-cell 2-1 selector on the Cell Matrix.   ∎

Exercise 3.2 Design a T flip flop on the Cell Matrix.   ∎

Exercise 3.3   Design a T flip flop without using D flip flops. This can be done with only 3 cells.   ∎

Exercise 3.4   Design a 4 bit register.   ∎

Exercise 3.5   Design a 4 bit register using only 5 cells (note that the clock will need to be sent to multiple cells, and the wires for routing the clock signal are not included in the 5 cells).   ∎

Exercise 3.6   Design a single cell for swapping the truth tables of two neighbors.   ∎

Exercise 3.7   Design a circuit on the Songline Processor for measuring how similar two signals are to each other.   ∎

Exercise 3.8   Design an SLP circuit for mathematically composing two functions.   ∎

Exercise 3.9   Design an SLP circuit for composing two functions, and programming a cell with that new composite function.   ∎

4. Enhancing the ZBC
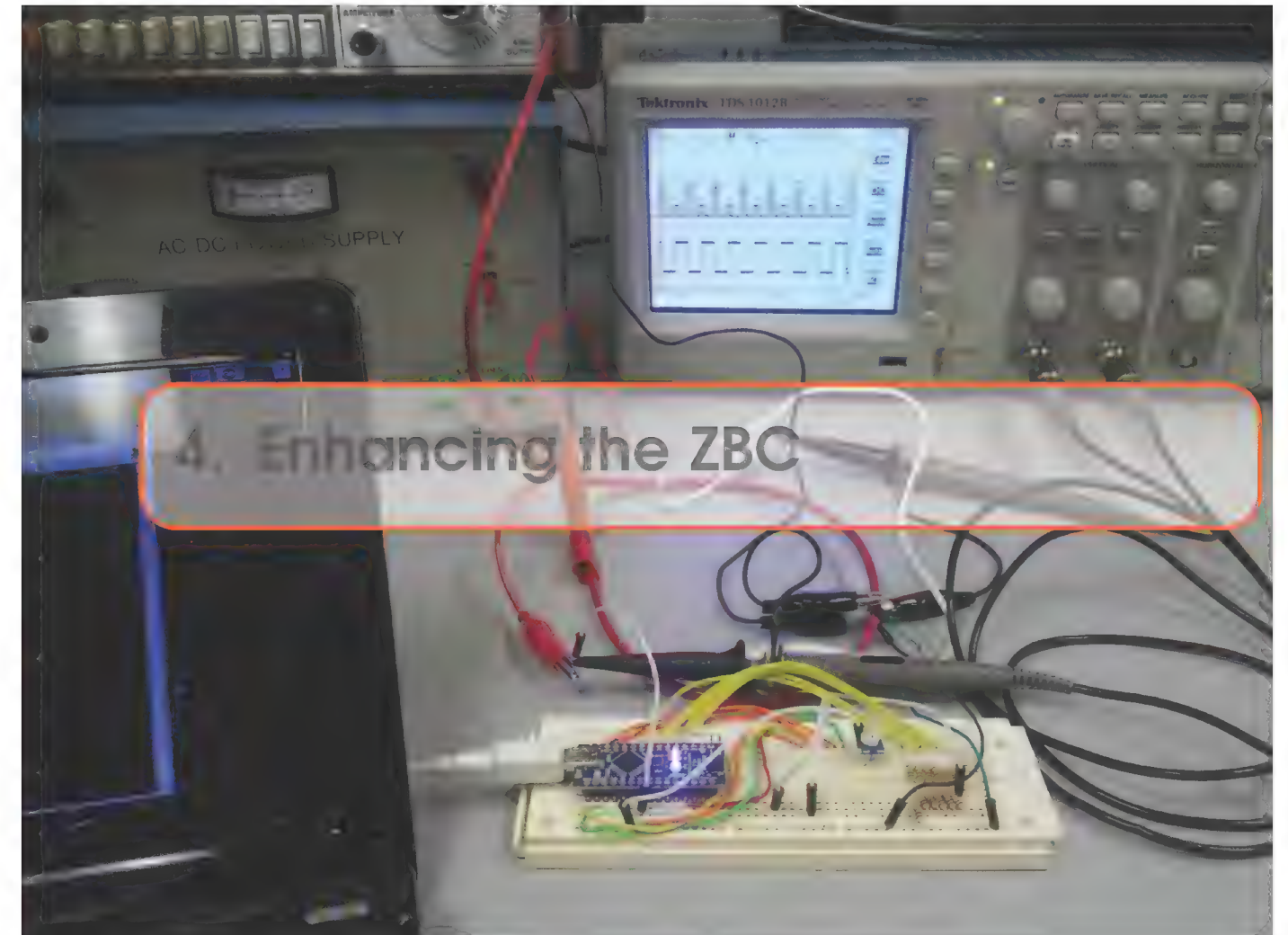
Recall that the ZBC is basically a transfer machine, programmed by naming pairs of SRC/DST addresses. The machine repeatedly performs copies from $SRC \rightarrow DST$, but despite this perhaps peculiar architecture, the system is still essentially a von Neumann machine: it has a program counter, which is used to pull instructions from memory, which perform read/modify/write operations on the memory. The PC is automatically incremented from instruction to instruction, but can also be explicitly loaded.

4.1   Removing the Ego

This architecture (along with most stored-program computers) already contains one important aspect of the Cell Matrix: the interchangeability of code and data. The contents of memory can be interpreted as data or as instructions, and in general, it is impossible from simply looking at the contents itself to tell whether that contents is data or code. We wish to preserve this interchangeability (this is the "egoless" aspect of EEXIST).

Despite this interchangeability, there is still an imperfection in this non-dualism. While a cell within the Cell Matrix can operate in *either* C or D mode, it is, at any given time,

in only one of those modes. It's not possible for a cell to be in *both* C and D mode. The Songline Processor makes some progress in this are, as its cells' C inputs accepts a real-valued number, allowing a cell to operate in a partial-C/partial-D mode. This is still imperfect: perfect non-dualism would be something where every part of the system is simultaneously acting-on and being acted-upon every part of the system. The shortcoming is not really in the architecture *per se*, but rather in the fact that instructions operate one at a time, so that (as with the Cell Matrix), some piece of memory may be a piece of code, but then may later be treated as a piece of data, and then still later as again a piece of code. This is our first hint that sequential execution will need to be abandoned.

4.2 Continuity of Time

Both the Cell Matrix and the Songline Processor have a schism in how they process information: in (pure) D-mode, everything happens asynchronously. Inputs cause outputs to change immediately, and those outputs flow into connected inputs, and so on. All this happens without a clock. But in C-mode, there is a system-wide clock which is used to control the sampling and presentation of inputs and outputs representing the new and old code. If we are to perfectly merge C and D mode so that every piece of the system is at all times (potentially) both a subject and object of a transfer, we will need to decide whether everything is clocked or everything operates in a dataflow mode. Using the natural world as a model, the decision was made to have the entire system operate without a clock. This however makes the notion of a PC mostly unusable. Consider a typical description of the PC: "the PC is incremented at the beginning of an instruction's execution cycle." This description is very clock-centric: the notions of "beginning" and "execution cycle" each convey a sense of clocked operation.

But if the PC is eliminated, what determines which instruction is being executed? The answer is simple: *all instructions are executed simultaneously*. This is perhaps a difficult thing to imagine. In the simple ZBC, each instruction specifies a transfer from $SRC \rightarrow DST$, and this transfer is an atomic unit: it begins, the transfer takes place, and then the instruction is finished. Rather than viewing this as clocked, we may view it as *discretized*. Following that line of thought, envisioning this as a clockless process means viewing the transfer as a *continuous* process. Thus, we imagine that the transfer from SRC to DST happens over a period of time [1], as opposed to at a single instant (e.g. the edge of a clock tick).

4.3 Continuity of Space

The most significant difference between the Cell Matrix and the Songline Processor is the change from discrete values (1 or 0) to continuous values (coming from the set of real numbers between 1 and 0). This continuity certainly makes modeling and interfacing with

---

[1]"clockless" is very different from "timeless." We still expect there to be a time component to operation of EEXIST.

with real world seem more natural, and, being in some sense a superset of the discrete version, seems reasonable to apply to the ZBC.

At first glance, this seems straightforward: let the values stored in memory be real-valued, so that transferring from, say $MEM[100] \rightarrow MEM[105]$ copies a real value instead of just an integer. So if $MEM[100]$ contained the value π then now so does $MEM[105]$ But because code and data are identical, what happens when the instruction pair stored at locations 100 and 101 is executed? That would imply a transfer from $MEM[\pi]$, whatever that means...

Here again, the real issue is discretization , in particular discretization of space. The conclusion is clear: *space* must also be made continuous. In particular, the address space in which instructions are stored should be addressable with real values. This is a very different kind of memory system from a typical von Neumann machine, as well as from the Cell Matrix and the Songline Processor.

4.4 Extended Effects: Karma (κ)

Assuming that a (theoretical) memory can be designed to be addressed by real numbers, there is an additional issue that arises. Suppose that (just for the sake of argument) an instruction specifies a transfer: $MEM[1.414] \rightarrow MEM[2.71828]$ In the context described so far, the effect of this instruction is like a *Dirac delta function*, in that it specifies a transfer from precisely one point to another. For example, while the data at location 2.71828 would be loaded with the contents of location 1.414, the data at location 2.7182800001 would be left unchanged, as would the data at 2.7182799999.

This seems a bit unnatural, at least in terms of everyday experience. It feels discontinuous, borderline chaotic: because if the instruction at $MEM[2.7182799999]$ is unrelated to the instruction at $MEM[2.7182800000]$, a slight perturbation in the coding of $MEM[1.414] \rightarrow MEM[2.71828]$ could have a significant change on the system's behavior.

In response to these notions, the idea of an *extended effect* is introduced. Basically, each transfer instruction $MEM[SRC] \rightarrow MEM[DST]$ is interpreted as not only requesting that data be copied from SRC to DST, but also from $SRC - \delta$ to $DST - \delta$, where δ ranges over some interval. However, the *strength* of that transfer weakens as $|\delta|$ increases. The exact meaning of "strength" will be defined in the next chapter. For now, suffice to say that not all transfers have the same impact on memory: some will barely change the contents of $MEM[DST]$, while others may impact the contents significantly.

This distributed impact is termed *Karma* (κ), and basically relates to the impact a transfer instruction may have on itself. Suppose, for example, $MEM[10]$ contains the instruction $MEM[20] \rightarrow MEM[11]$. The instruction at location 10 is requesting data be copied from location 20 to location 11. Normally, only the contents of $MEM[11]$ would be affected by this. In the presence of karma, locations near address 11 would also be affected. If κ is large enough, then $MEM[10]$ would also be affected, i.e., the instruction's requested action is impacting itself. In other words, there is a mixing of effect with cause .

This completes the basic catalog of characteristics we wish to have in the enhanced ZBC. What is required next is a model for implementing these characteristics. The chosen

model is not necessarily practical, but will give us a simulation target with which we can explore this system. This model is called "*EEXIST*": **E**goless **EX**tended effect cont**I**nuous **S**pace **T**ime .
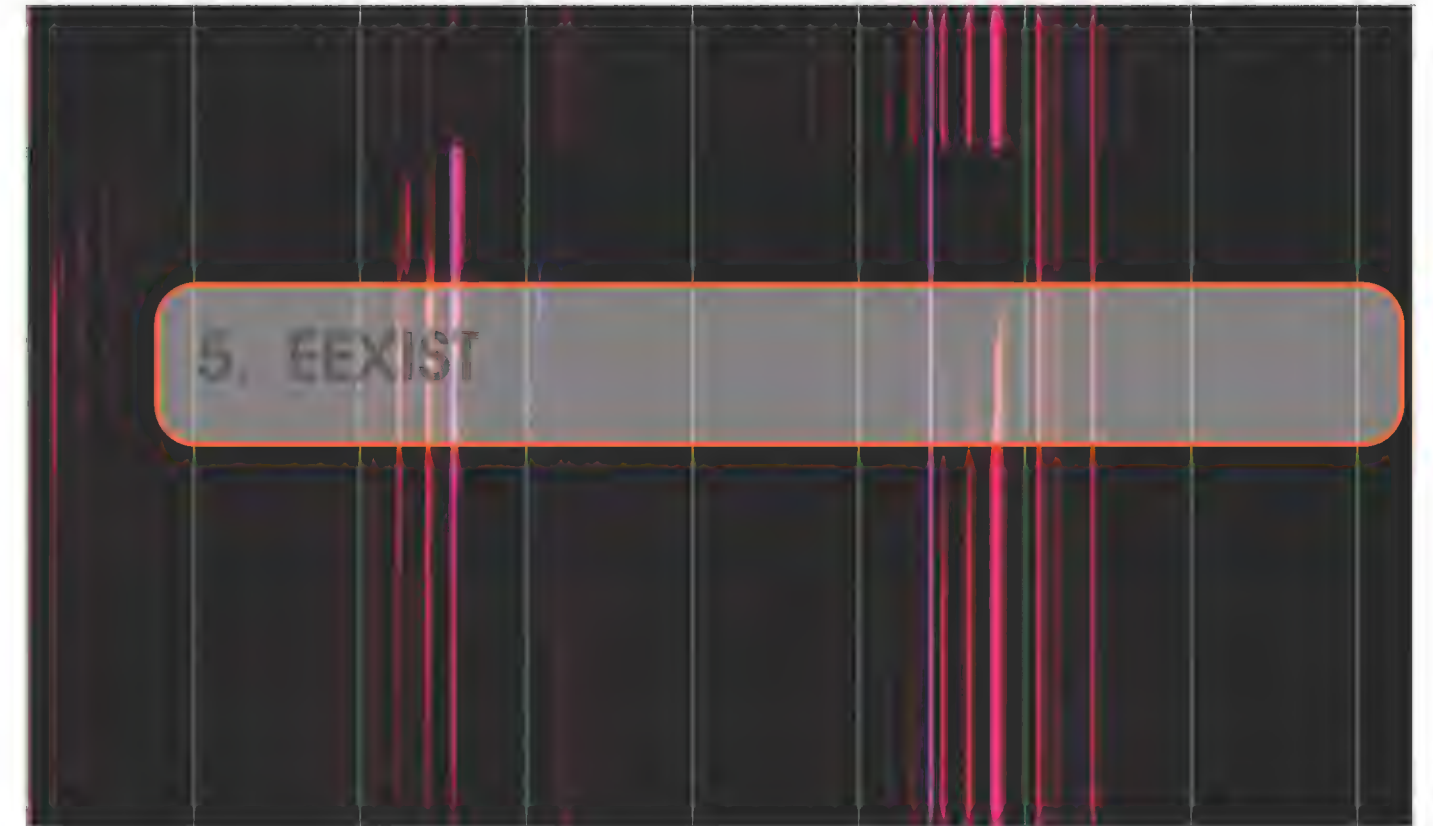
Exploring this proposed architecture – studying its behavior, potential applications, and so on – requires some sort of model that captures the characteristics described above. The proposed model ("EEXIST") is neither unique nor necessarily ideal, but it has worked well enough to begin exploring these concepts.

5.1 Memory Structure

A traditional memory subsystem, viewed as a collection of indexed holding bins, does not extend to a spatially-continuous layout as needed for EEXIST. Instead, imagine a porous substrate, capable of holding liquid at any place throughout its extent: something like a long, rectangular sponge with its largest dimension labeled "x" (figure 5.1). The amount of liquid present at some location x would represent the value stored at that address , i.e. $MEM[x]$. Let's clarify the notion of "amount of liquid": if x is truly a point location, then the em volume of liquid at x tends towards 0 (as $\delta x \to 0$ in our volume calculation). But consider, instead, the *height* of the liquid at address x, as shown in figure 5.1.

The contents of $MEM[x]$ is now simply the height of the liquid at position x along the horizontal dimension. Not that while this appears to change smoothly in the figure, no assumption is made about continuity of values.
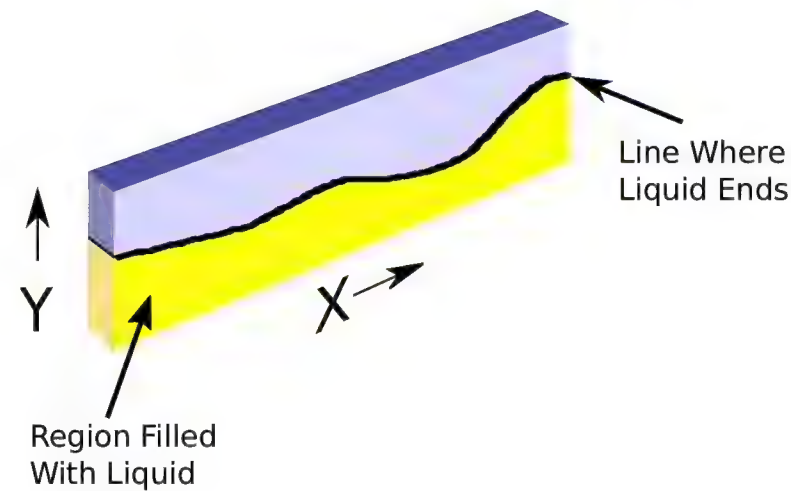
Figure 5.1: Image of EEXIST Memory as a Liquid-Filled Sponge. The line shows the height of the liquid at different x positions.

Since memory will be addressed by a continuum of values, there is no longer a notion of "next" or "adjacent" addresses: we cannot treat memory addresses in pairs as we can with a normal memory. So instead of coding $SRC \to DST$ instructions in pairs of addresses, we store one transfer instruction per address. To do this, we consider two different liquids, named "SRC" and "DST." Throughout, SRC will be show as a red chemical , and DST as a blue chemical. At any given position x in the medium, the mixing of these chemicals is irrelevant, as is their location along the y dimension. For simplicity, we will usually show the SRC chemical as situated below the DST chemical; but all that is relevant is the height of the regions containing the SRC and DST chemicals at a given x location.

Figure 5.2 shows a front view of this memory setup. The lower red region shows the values of SRC, while the upper blue region shows DST. For the position marked x, it is the thickness of the SRC region that codes the actual SRC address. Similarly, the thickness of the DST region codes the DST address. In the figure, at the point $x = 24.5$, we have a SRC value of 2.1 and a DST value of 1.7. Hence, $MEM[24.5]$ codes a transfer instruction $MEM[2.1] \to MEM[1.7]$.

5.2   Discretization of Memory

In this example, since SRC and DST vary smoothly, the instruction at, say, $MEM[4.51]$ would be similar to what's at $MEM[4.5]$. While there is no requirement for continuity in the boundaries of the SRC and DST regions, having these change smoothly allows us to approximate the system with a series of discrete tubes [1], as shown in figure 5.3. Here we are breaking the x dimension into a series of thin regions, where each region has a given

---

[1]This discretization does not contradict our original motivation for making the spatial dimension continuous: it's merely a nod to ease of simulation.
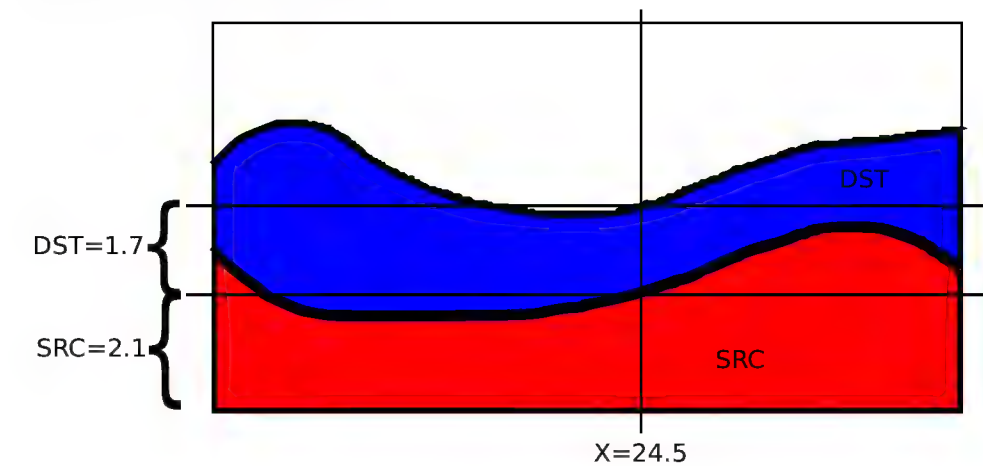
Figure 5.2: Front-view of the Chemical Memory. The red region corresponds to the SRC chemical, and the blue region corresponds to the DST chemical. The relative position of SRC and DST (e.g. which is on top) is irrelevant: the height *of each region (top to bottom) codes the desired address.*

height of SRC and DST chemicals (where again, SRC and DST are color coded red and blue, respectively).
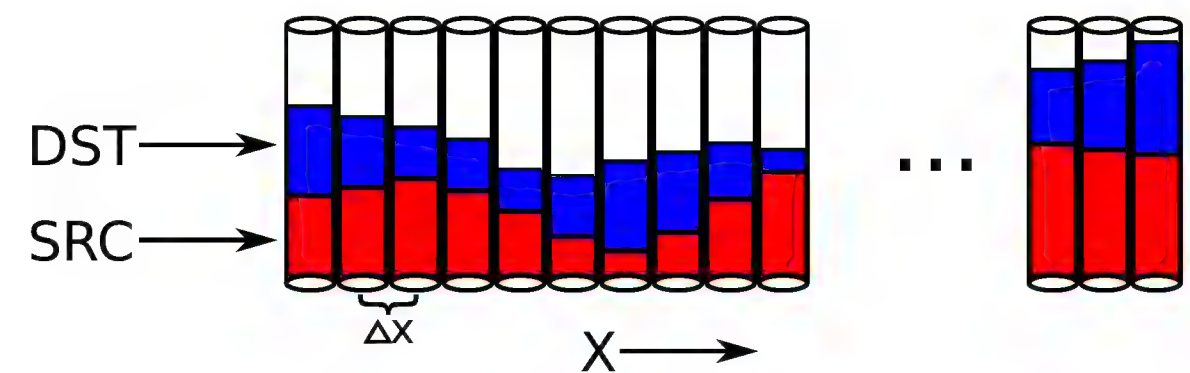

Figure 5.3: Discretized View of SRC/DST Memory. The x dimension is broken into a set of small intervals, each viewed as a thin tube containing both SRC and DST chemicals. SRC is shown in red, DST is shown in blue.

This is only an approximation of an ideal EEXIST memory; but if we let these tubes grow thinner and more numerous, then this model approaches the ideal. This discretized model makes it easier to simulate the system though, and is the model we'll work with throughout this text. It remains an open question whether this model changes smoothly to the ideal in the limit, or if the behavior of the system somehow fundamentally changes in the pure-continuous case. See Part III for this and other discussions of future-work.

In this context, for example, a memory transfer instruction such as $MEM[5] \to MEM[6]$ would request that the contents of memory at location $x = 5$ be transferred to location

$x = 6$. This differs from a usual memory copy in two ways:

1. The instruction actually specifies a *transfer* of contents rather than a simple copy. In EEXIST, chemicals (both SRC and DST chemicals) are moved from the SRC location to the DST location; and
2. the transfer is not instantaneous, but rather occurs over a period of time.

If the instruction executes for enough time, eventually a complete transfer will have taken place, and $MEM[SRC]$ will be empty, its contents completely moved to $MEM[DST]$. In practice, this doesn't happen, for two reasons:

1. The transfer instruction itself is likely to change, in response to other transfer requests elsewhere in the system (or possibly in response to itself); and
2. other transfer requests may continually be loading chemicals *into MEM[SRC]*.

5.3   Extended Transfer Effects: Karma (κ)

Viewing the memory of EEXIST in this way, transfer instructions themselves can be coded precisely (since the level of SRC and DST chemicals are real-valued), but the *action* of those transfers is approximated. For example, suppose the x domain ranges from 0 to 40 (inclusive), and is broken into tubes of width 0.03125. Suppose also $MEM[0]$ specifies a transfer $MEM[5.51] \rightarrow MEM[6.02]$. There are tubes corresponding to 5.50000 and 5.53125, but not to exactly 5.51. Similarly, there are tubes corresponding to 6.00000 and 6.03125, but not to exactly 6.02. This raises a question of how to map such a transfer to our discretized space.

Rather than just rounding transfer addresses to the nearest tube, EEXIST distributes the effect of a transfer request to a region of tubes. Continuing the above example of a transfer from $MEM[5.51] \rightarrow MEM[6.02]$, this instruction would be interpreted as transferring from a region of memory locations *centered at 5.51* to a region of memory locations centered at 6.02. The exact nature of these nearby transfers and their diminished impact will be explained in detail in the *Mechanics* section later in this chapter.

Figure 5.4 shows a profile of how much a transfer request from $MEM[5.51]$ will affect nearby regions. Ignoring discretization of space, such an instruction will have the greatest impact on $MEM[5.51]$, but will also have a (lesser) impact on $MEM[5.50]$ and $MEM[5.52]$. The impact will also affect (to an even lesser degree) $MEM[5.49]$ and $MEM[5.53]$, and so on. The impact curve could be made bell-shaped, but for simplicity and speed of simulation, a simple linear profile has been chosen.

κ controls the extent of this distributed effect (the "karma" of the system). A transfer request $MEM[x] \rightarrow MEM[y]$ will actually affect tubes from $MEM[x - \kappa]$ to $MEM[x + \kappa]$, transferring chemicals from those tubes to tubes between $MEM[y - \kappa]$ and $MEM[y + \kappa]$. The most pronounced effect will be at $MEM[x]$ and $MEM[y]$, with the effect tapering off to zero at e.g. $MEM[x - \kappa]$ and $MEM[y - \kappa]$. This turns out to be useful far beyond the simple question of mapping in discretized space: the karma of the system fundamentally affects the large-scale behavior of the system.

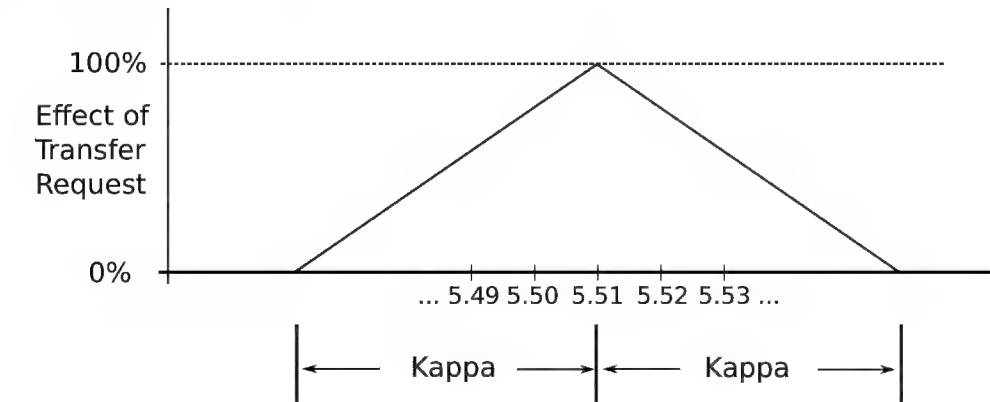The extreme values for κ are worth considering:

Figure 5.4: Curve Showing Effect of a Transfer Request. The most prominent effect will be centered at $MEM[5.51]$, but nearby locations will also be affected, with the effect dropping off linearly for a distance of κ on either side of 5.51. For a transfer request centered at location p, the effect at location x is denoted $D_\kappa(x, p)$
.

- If $\kappa = 0$, transfers have a purely local effect. Each transfer specifies a point-source and point-destination. This in some ways mimics the behavior of a traditional (discretely-address) memory.
- If $\kappa = X_{max}$, then any transfer will affect *every* region of the memory, though the effect diminishes away from the specified SRC and DST.
- If $\kappa = \infty$, then any transfer affects all regions equally, i.e. a transfer request from
$MEM[SRC] \rightarrow MEM[DST]$
causes an equivalent transfer
$MEM[SRC - \Delta] \rightarrow MEM[DST - \Delta]$
for all Δ.

κ is generally fixed for the entire system, but in some experiments it has been modified over the course of a run. It could also theoretically change from spatial point to point.

5.4   Discretization in Time

To allow for transfer requests to interact with one another, chemical transfers do not occur instantaneously, but rather over a period of time. For example, the transfer request $MEM[5] \rightarrow MEM[10]$ says to move chemicals from location 5 to location 10. One may view this as connecting a pipe between those locations, and allowing chemicals to be siphoned off from location 5 to location 10. This transfer happens at a finite rate, so that over time the chemical level at $MEM[5]$ drops while the level at $MEM[10]$ rises. The details of this transfer are described below in the *Mechanics* section.

Such a transfer can be approximating by repeatedly incrementing time by a small amount Δt, where $\Delta t > 0$. The goal of course is to have Δt approach 0, but in practical terms, the smaller Δt is, the longer the simulation takes. If we imagine Δx and Δt approaching 0,

we can describe the state of the system with a pair of equations.

We idealize κ (karma) by using a curve for the effect $D_\kappa(x,p)$ (the "effective diameter due to κ") on $MEM[x]$ of a transfer request at location p, and letting $D_\kappa(x,p) = e^{-\frac{(x-p)^2}{\kappa}}$. If $\kappa = 0$ then $D_\kappa(x,p)$ is 0 everywhere except where $x = p$. As κ increases, the effect $D_\kappa(x,p)$ still has a maximum value where $x = p$, and tapers-off (but is non-zero) everywhere else. As $\kappa \to \infty$, $D_\kappa(x,p)$ flattens out, and in the limit, is equal to 1 everywhere. See figure 5.5.

(a)

(b)

(c)

(d)
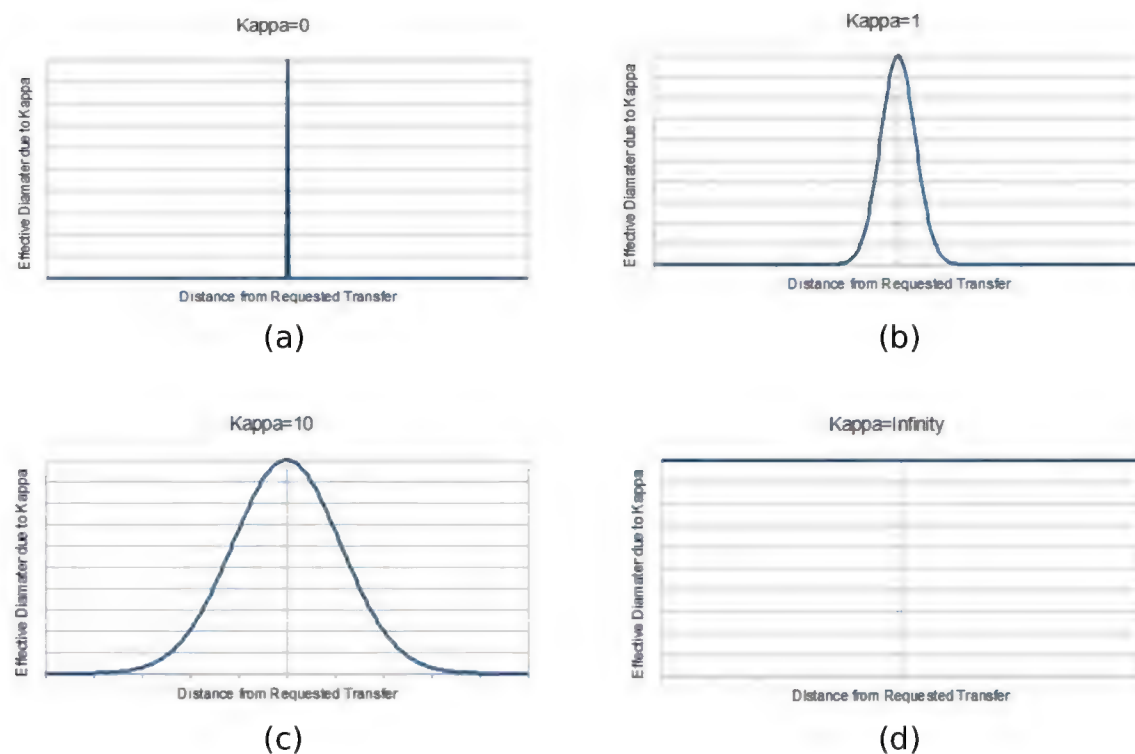
Figure 5.5: Effect of κ on $D_\kappa(x,p)$. For $\kappa = 0$, this is an impulse function (a). As κ increases, it becomes a bell-shaped curve (b), distributing transfer effects over more and more space. As κ approaches ∞, the curve widens (c) until it is a flat line (d).

Let $SRC(p,T)$ and $DST(p,T)$ be the amount of SRC and DST chemicals (respectively) at position p and time T. We can now describe the theoretical behavior of SRC and DST with a pair of integral equations:

$$SRC(p,T) = SRC(p,0) +$$

$$\int_0^T \int_{-\infty}^\infty e^{-\frac{(DST(x,t)-p)^2}{\kappa}} - e^{-\frac{(SRC(x,t)-p)^2}{\kappa}} dxdt$$

$$DST(p,T) = DST(p,0) -$$

$$\int_0^T \int_{-\infty}^\infty e^{-\frac{(DST(x,t)-p)^2}{\kappa}} - e^{-\frac{(SRC(x,t)-p)^2}{\kappa}} dxdt$$

5.5 Simulation Mechanics

Simulation of EEXIST has changed little since the beginning of this project. This doesn't suggest that the chosen knob settings are ideal; rather, it shows (perhaps) the insensitivity of the system to the particular choice of settings.

Current settings are as follows:

- range of x: 0.0 to 40.0
- Δx: 0.03125
- Δt: 0.05
- κ: typically 5, but sometimes between 2 and 10

If a transfer instruction requests a transfer to (or from) location p, the effect on $MEM[x]$ is defined by $D_\kappa(x,p)$, which is approximated with a set of linear functions, as follows:

$$D_\kappa(x,p) = \begin{cases} 0, & \text{if } x \leq p - \kappa \\ \frac{\kappa - |p-x|}{\kappa}, & \text{if } p - \kappa < x < p + \kappa \\ 0, & \text{if } x \geq p + \kappa \end{cases} \tag{5.1}$$

5.5.1 Transfer Addressing

The first step in simulating the effect of transfer requests is to determine the actual source and destination addresses for each transfer. Suppose $MEM[X]$ contains the instruction: $MEM[SRC] \to MEM[DST]$. This can be interpreted two ways:

1. SRC and DST can be the *absolute* source and destination addresses; or
2. SRC and DST $X + SRC$ and $X + DST$ can be *relative* addresses, i.e., $X + SRC$ and $X + DST$ are the actual addresses.

The choice of relative or absolute addressing is selectable in the EEXIST API. It turns out though this distinction is immaterial: the introduction of *bias* (discussed in the next section) allows the emulation of relative addressing using absolute addressing.

5.5.2 Transfer Type

Karma (and $D_\kappa(x,p)$) and the transfer type determine the final SRC and DST addresses of a transfer. Once those have been determined, there are two ways in which the actual transfer can take place:

1. "SD flow," which specifies a flow rate form SRC to DST based on the *amount* of SRC chemical; and
2. "Equilibrium flow," which modulates the flow rate based on the *difference* in the amount of chemicals at SRC and DST.

The main difference is that SD flow will eventually drain the SRC location, whereas equilibrium flow will eventually cause the chemical levels at SRC and DST to be the same. The choice of transfer type is selectable in the EEXIST API, though almost all work to date has been done under an assumption of SD Flow.

5.5.3 Main Simulation Loop

The basic simulation update loop proceeds as follows:

1. Look at each transfer instruction from $p = 0$ to $p = 40$; suppose $MEM[p] : MEM[SRC] \rightarrow MEM[DST]$;
2. calculate the corresponding change to $MEM[SRC]$ and $MEM[DST]$:
   - $\Delta MEM[SRC] = -MEM[SRC] * D_\kappa(x, p) * \Delta t * \Delta x$
   - $\Delta MEM[DST] = MEM[SRC] * D_\kappa(x, p) * \Delta t * \Delta x$

   For equilibrium flow, the equations are:
   - $\Delta MEM[SRC] = -(MEM[SRC] - MEM[DST]) * D_\kappa(x, p) * \Delta t * \Delta x$
   - $\Delta MEM[DST] = (MEM[SRC] - MEM[DST]) * D_\kappa(x, p) * \Delta t * \Delta x$
3. accumulate all these Δ's, **being careful not to let any amounts go negative**;
4. after evaluating all transfer instruction, apply the accumulated Δ's to each memory location.

Note that each Δ calculated above (for example, $\Delta MEM[SRC]$) describes *a pair* of changes: a change to the amount of SRC chemical at $MEM[SRC]$ and a change to the amount of DST chemical at $MEM[SRC]$.

Figure 5.6 (a) shows an example of an absolute SD transfer operation. The instruction at $MEM[5]$ specifies the transfer $MEM[2] \rightarrow MEM[10]$. Assuming for simplicity $\kappa = 0$ (so $D_\kappa(x, p) = 1 \forall x, p$), $\Delta x = .1$ and $\Delta t = .1$, this single instruction requests a transfer of $8 * 1 * .1 * .1 = .08$ SRC from $MEM[2]$ to $MEM[10]$, and a transfer of $9 * 1 * .1 * .1 = .09$ DST from $MEM[2]$ to $MEM[10]$, resulting in the concentrations shown in figure 5.6(b).

Note the following:

1. Figure 5.6 only shows the effect of the instruction at $MEM[5]$; the instructions at $MEM[2]$ and $MEM[10]$ also affect memory.
2. The instruction at $MEM[5]$ does not affect the contents of $MEM[5]$. The SRC and DST chemicals only specify where a transfer should take place; those chemicals are not directly affected by the transfer, unless the instruction (taking karma into account) refers to its own address.

Assuming absolute addressing and SD flow, this will transfer SRC and DST chemicals from location 2 to location 10. Under equilibrium flow, this would transfer SRC chemicals 2 to 10, and DST chemicals from 10 to 2. With relative addressing, the instruction requests transfers from 7 to 15 ($5 + 2$ to $5 + 10$).

5.6 Effect of Karma

It's interesting to look at the effect of changing κ on the behavior of the system. For this analysis, a configuration that implements a 3 input exclusive-or (XOR) gate was employed. The nature of the results shown here seem pretty general, regardless of the specific system
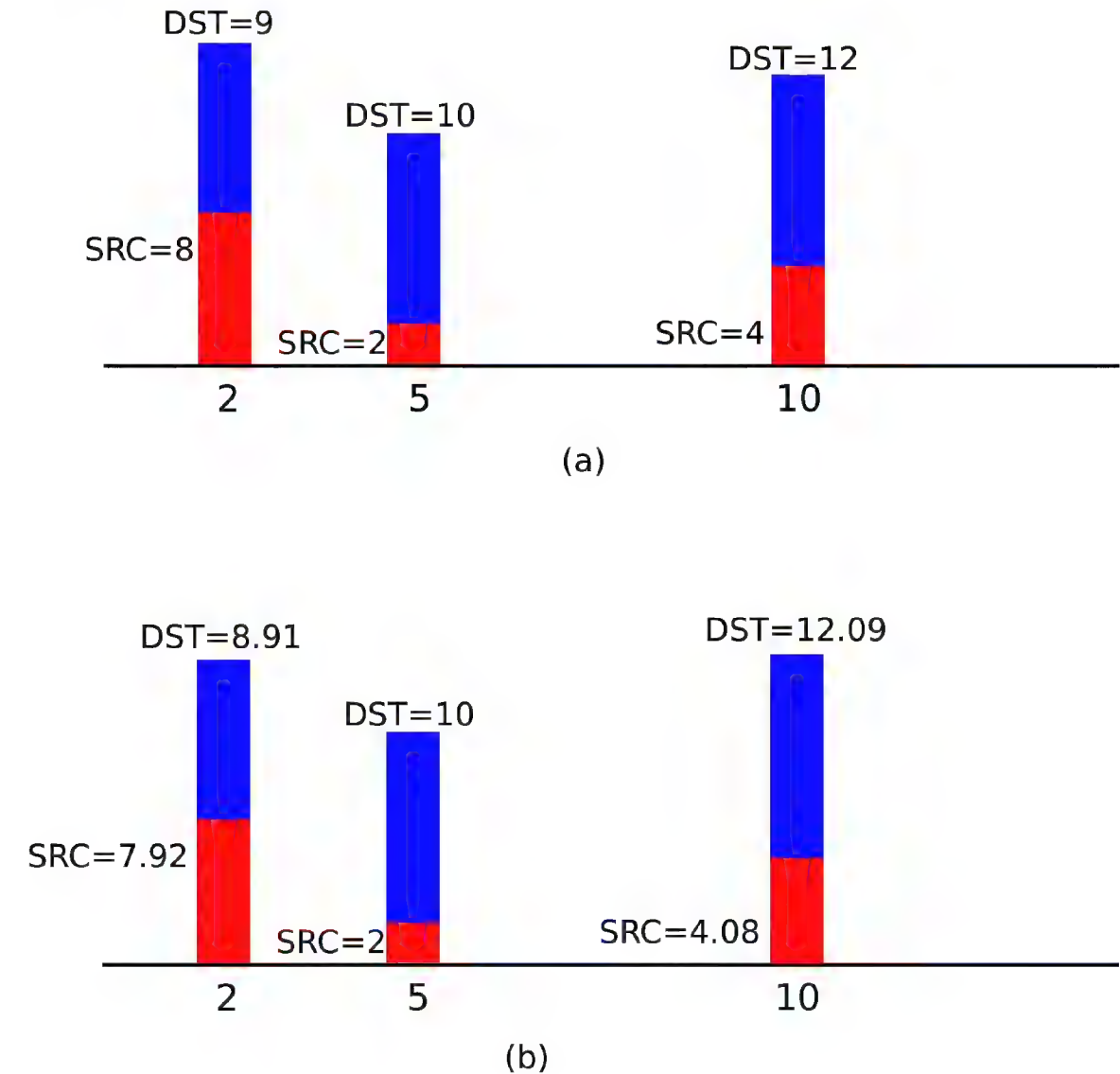
(a)

(b)

Figure 5.6: Sample Set of Transfer Requests. (a) shows an initial configuration, where $MEM[5]$ specifies a transfer from $MEM[2] \rightarrow MEM[10]$. (b) shows the results of that single transfer in absolute, SD flow mode with $\kappa = 0$. Locations 2 and 10 also specify transfer requests, but the effect of those are not shown.

configuration. Figures 5.7 - 5.18 show the chemical distribution in this system after a number of timesteps have passed.

A nominal value of $\kappa = 5$ has been used for most work to date. There's no particular reason for this choice; it was simply the initial choice, and it just hasn't changed. Figure 5.7 shows the chemical balance in the evolved XOR system. The distribution is stable: it does not change over time, unless the chemical balance is perturbed from outside the system.
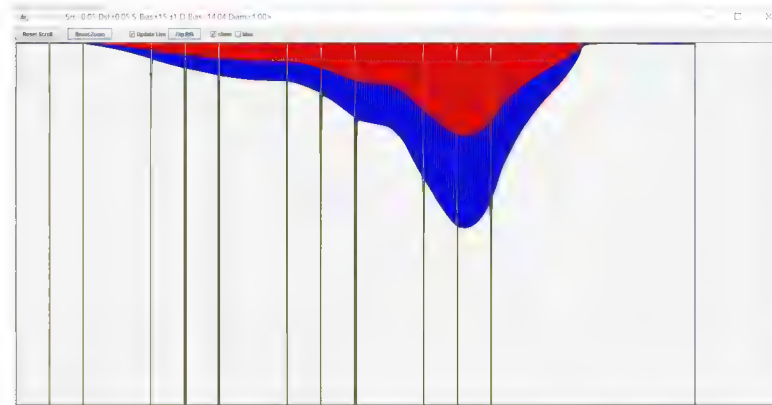
Figure 5.7: System Behavior With $\kappa = 5$. The chemical balance is stable, unchanging as the simulation is advanced.

In figure 5.8, κ has been increased to 6. The chemical balance changes, and the system stabilizes again.
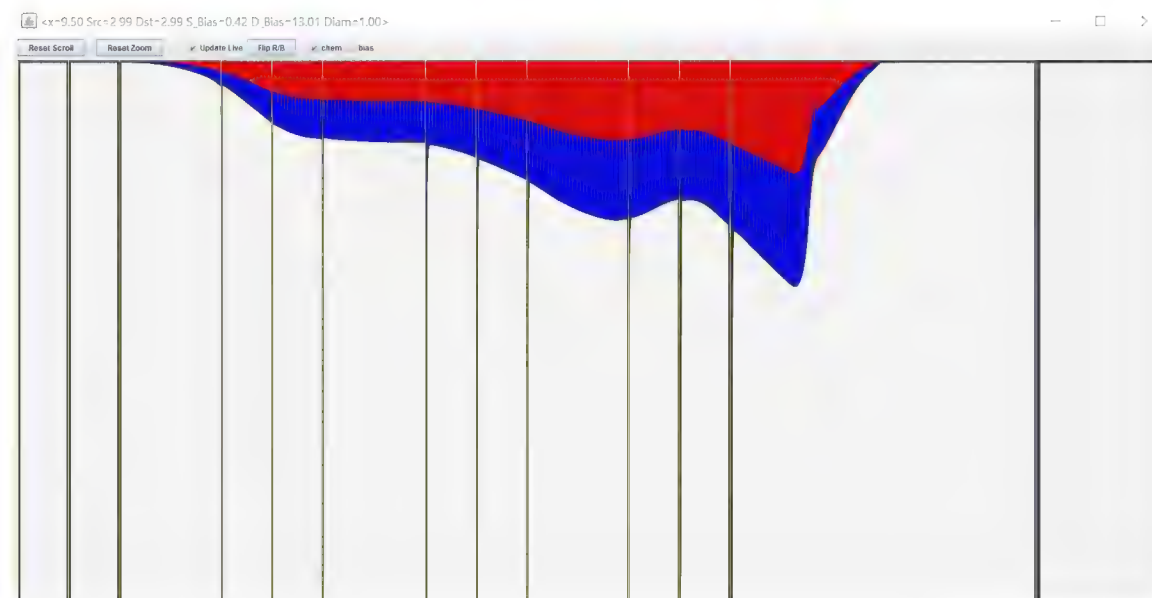

Figure 5.8: System Behavior With $\kappa = 6$. The chemical balance has changed from $\kappa = 5$ but has stabilized again.

Similar behavior is observed as κ changes to 8 (figure 5.9 and 15 (figure 5.10.

At $\kappa = 25$ (figure 5.11), the chemical distribution begins to flatten out. While still stable over time, a small perturbation can be seen near the left end of the system.

When κ increases to 27 (figure 5.12), the perturbation seen in figure 5.11 begins to change. Figure 5.12 shows two snapshots of the system at different points in time.
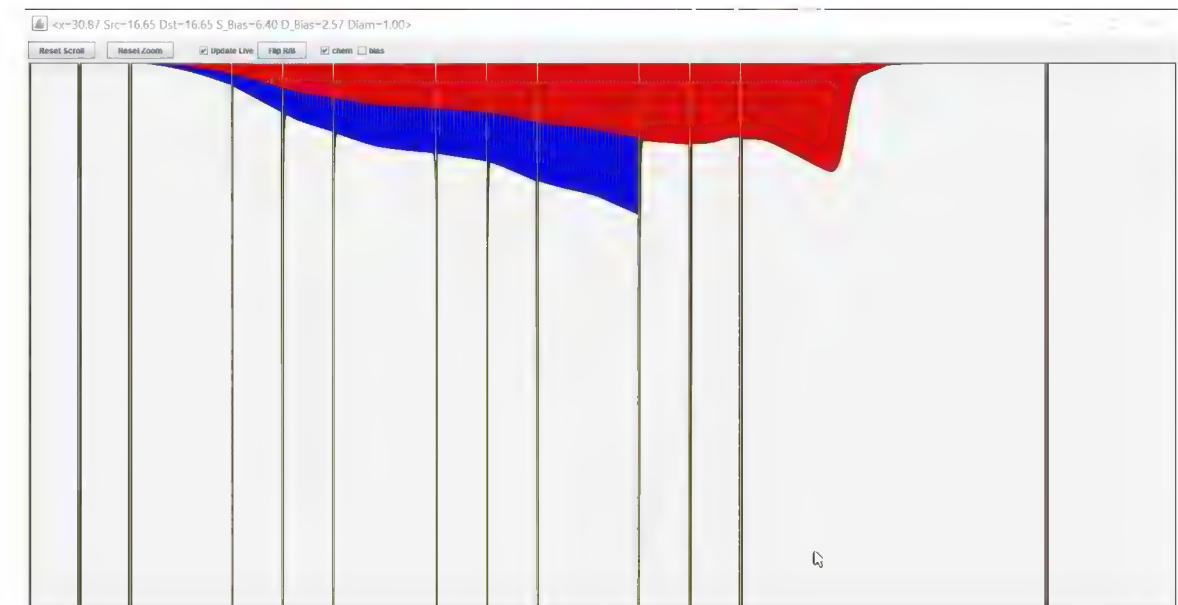
Figure 5.9: System Behavior With $\kappa = 8$.
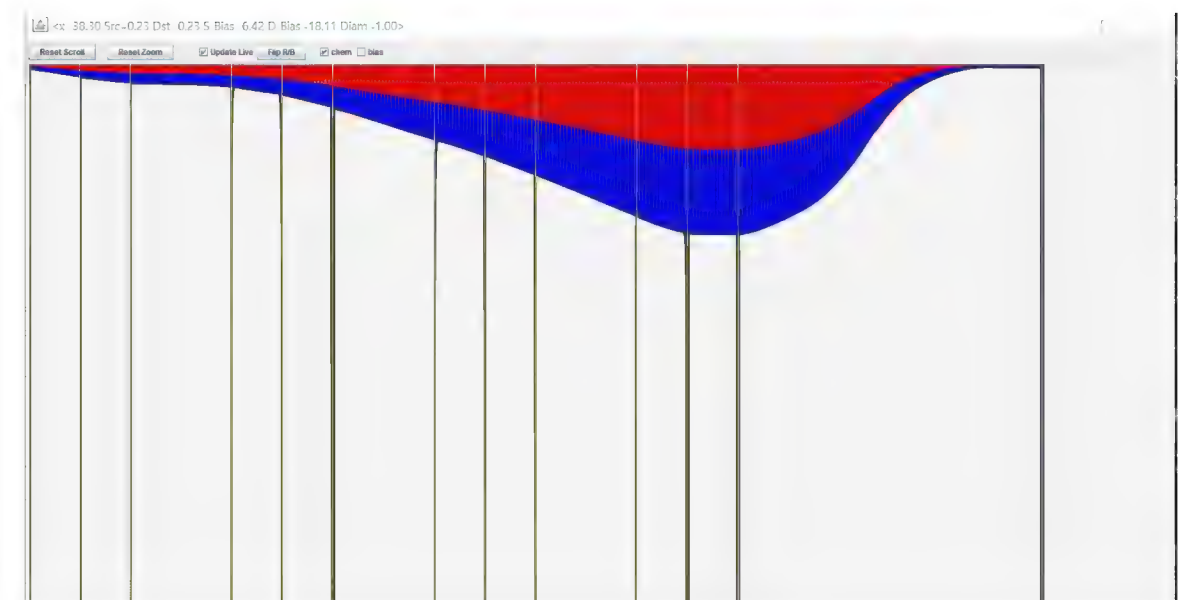
Figure 5.10: System Behavior With $\kappa = 15$.

At $\kappa = 30$ (figure 5.13), multiple small regions are changing between two states.

As κ continues to increase, the chemical balance changes to a series of peaks and troughs, which shift in position over time (figures 5.14 (a)-(c)).

Figure 5.15 shows the system when $\kappa = 40$. The behavior is the same as that shown in
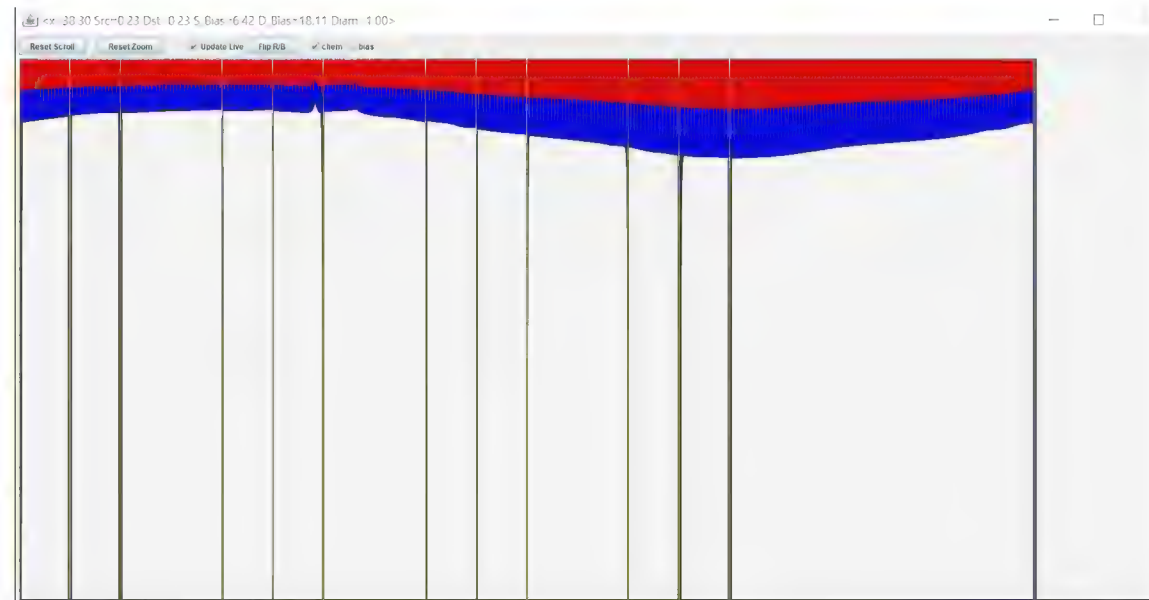
Figure 5.11: System Behavior With $\kappa = 25$. The chemical distribution seems to be flattening out, but a small perturbation appears near the left.

figure 5.14, but the shape of the peaks has become more uniform.

These behaviors are not entirely unexpected. The mechanics of EEXIST instruction interpretation basically link cause to effect; *karma (κ) allows those effects to come back and affect the cause*. This allows the creation of feedback, which one typically expects to lead to either stability (as seen when κ is between say 5 and 25) or oscillation ($\kappa > 25$).

It is also interesting to look at what happens as κ approaches 0. Figure 5.16 shows the system with $\kappa = 2$. Here, the chemical balance is beginning to change over time.

At $\kappa = 1$, the system looks non-periodic. Figure 5.17 (a)-(d) shows 4 snapshots of the system at different (non-successive) timesteps.

When $\kappa = 0$ the system changes fundamentally, and appears very disorganized and random (possibly chaotic?). Figure 5.18 (a)-(d) show 4 snapshots of the system, with no apparent structure to the chemical balance.

This behavior is also no entirely unexpected: without any feedback in the system, each instruction modified the system's chemical balance, but all instructions act entirely on their own, without any direct consequence on themselves. One might expect this to lead to a large number of seemingly unrelated changes, which is one way of interpreting figure 5.18.

5.7   Bias, Diameters

There are two useful embellishments to the above mechanisms for EEXIST. One is allowing an offset ("bias") to be applied to each location; the other is allowing specification of a
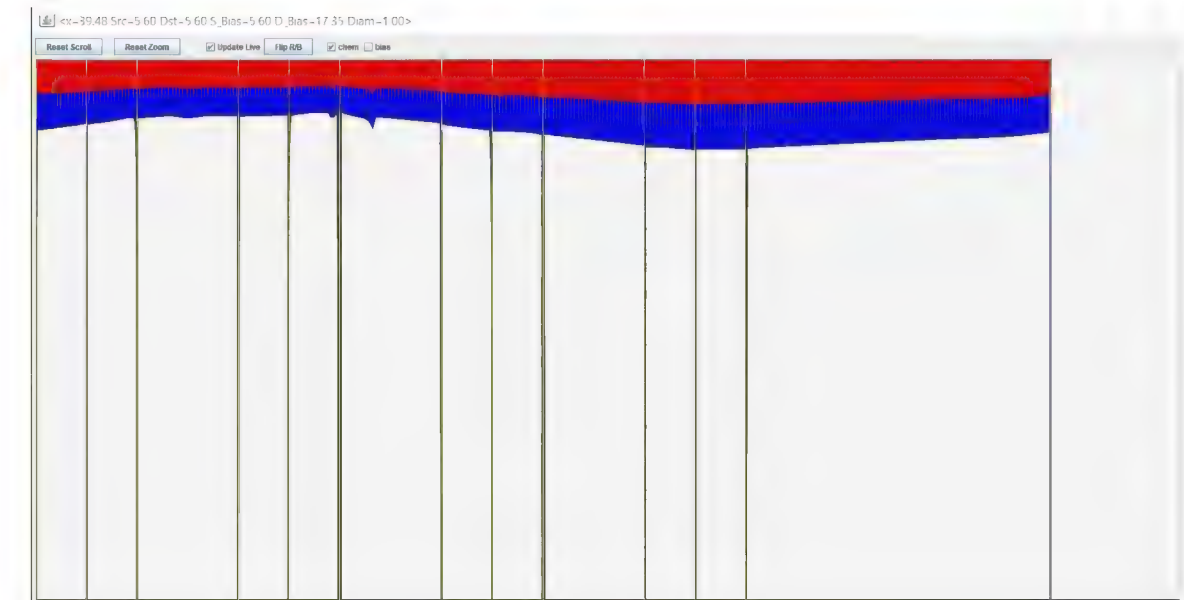
(a)

(b)

Figure 5.12: System Behavior With $\kappa = 27$. Figures (a) and (b) show two states between which the system alternates.

flow restriction ("diameter") to each location.
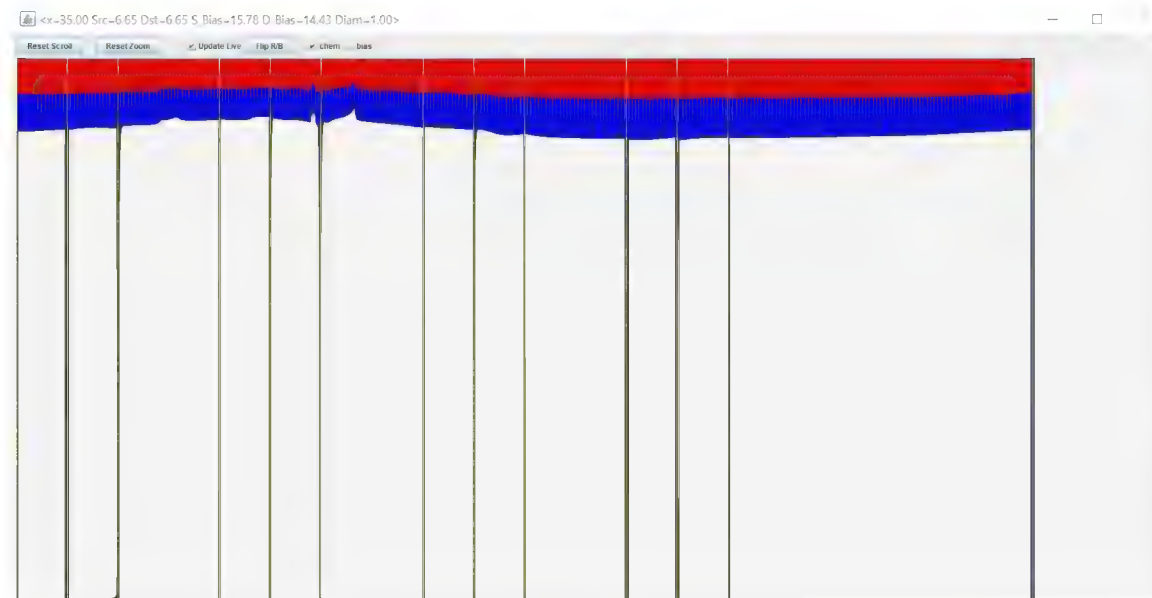
(a)

(b)

Figure 5.13: *System Behavior With* $\kappa = 30$. *Multiple regions are changing across time.*
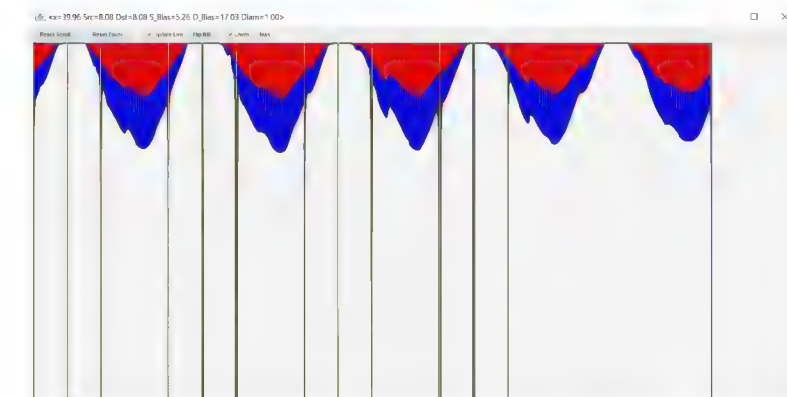
Bias is a set of pairs of offsets (Bias.SRC and Bias.DST) which are added to the SRC and DST locations specified by an instruction. Normally, $Bias(x) = 0 \forall x$, meaning an
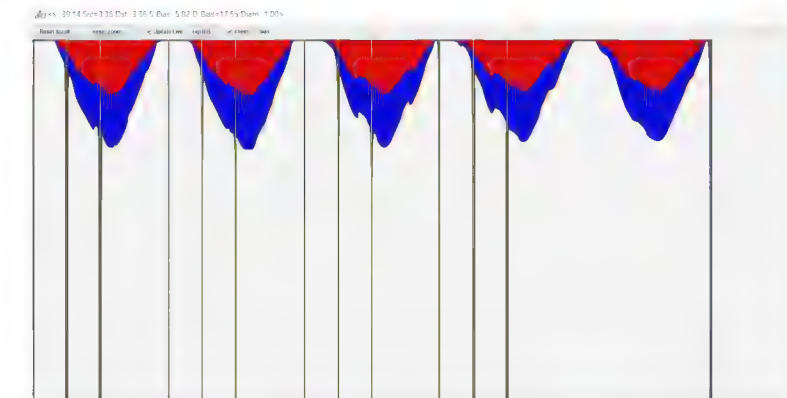
(a)
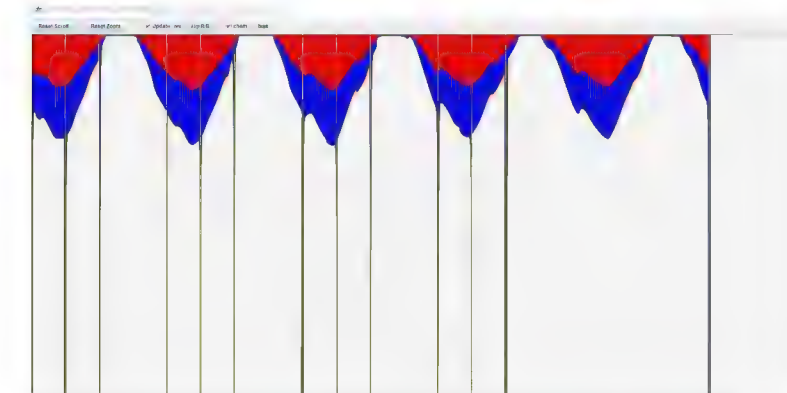
(b)

(c)

Figure 5.14: *System Behavior With* $\kappa = 35$. *The chemical balance shows a series of peaks and troughs, whose positions shift over time.*

instruction such as $MEM[2] \rightarrow MEM[10]$ refers to addresses (centered at) 2 and 10. If that instruction is itself stored at address 5 (as in figure 5.6 (a)), and $Bias.SRC(5) = 20$
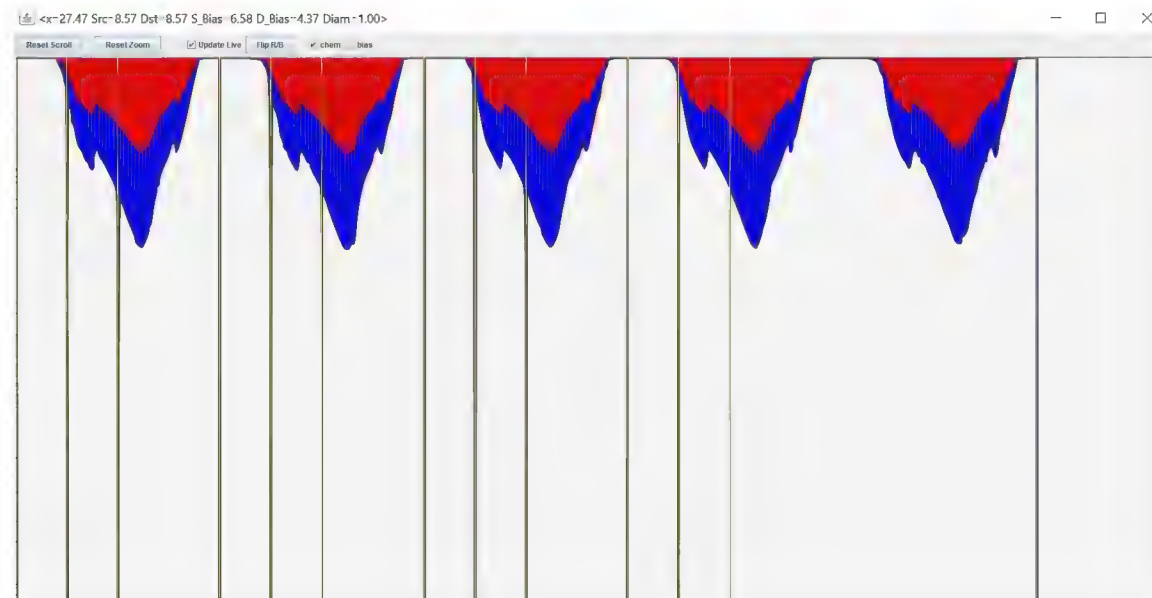
Figure 5.15: System Behavior With $\kappa = 40$.
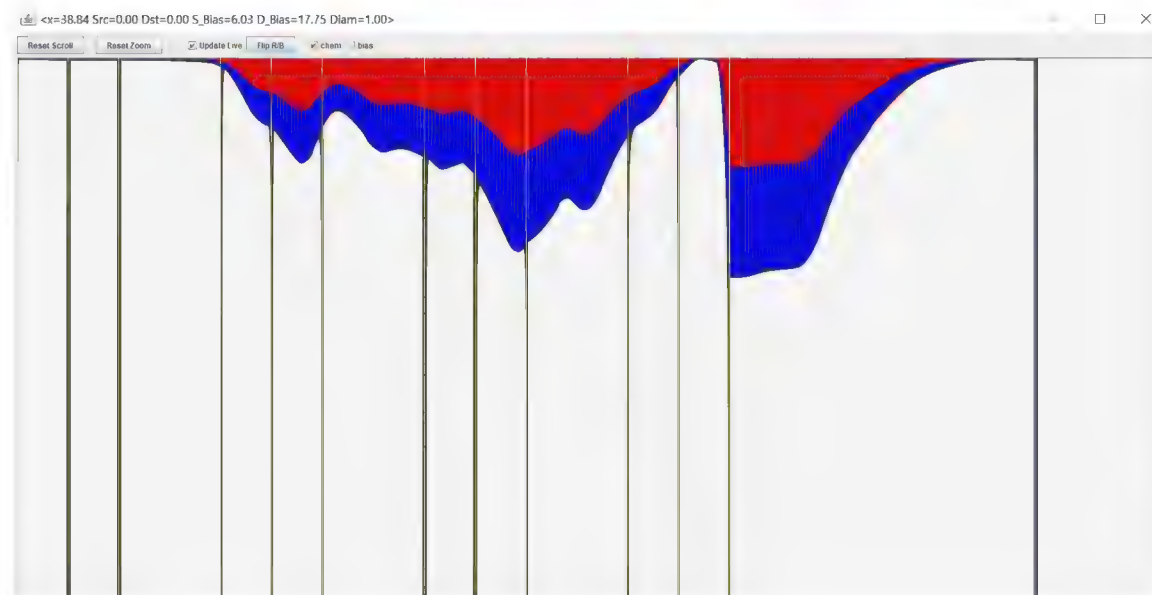
Figure 5.16: System Behavior With $\kappa = 2$. The chemical balance is changing over time.

and $Bias.DST(5) = -1$ then the instruction actually requests a transfer $MEM[2+20] \rightarrow MEM[10-1]$. As mentioned above, Bias can be used to emulate relative addressing in an absolute-addressed system: by setting $Bias(x) = x \forall x$, the SRC and DST specified by a transfer instruction are added to the address of the instruction, thereby acting as if they are

(a)

(b)

(c)

(d)
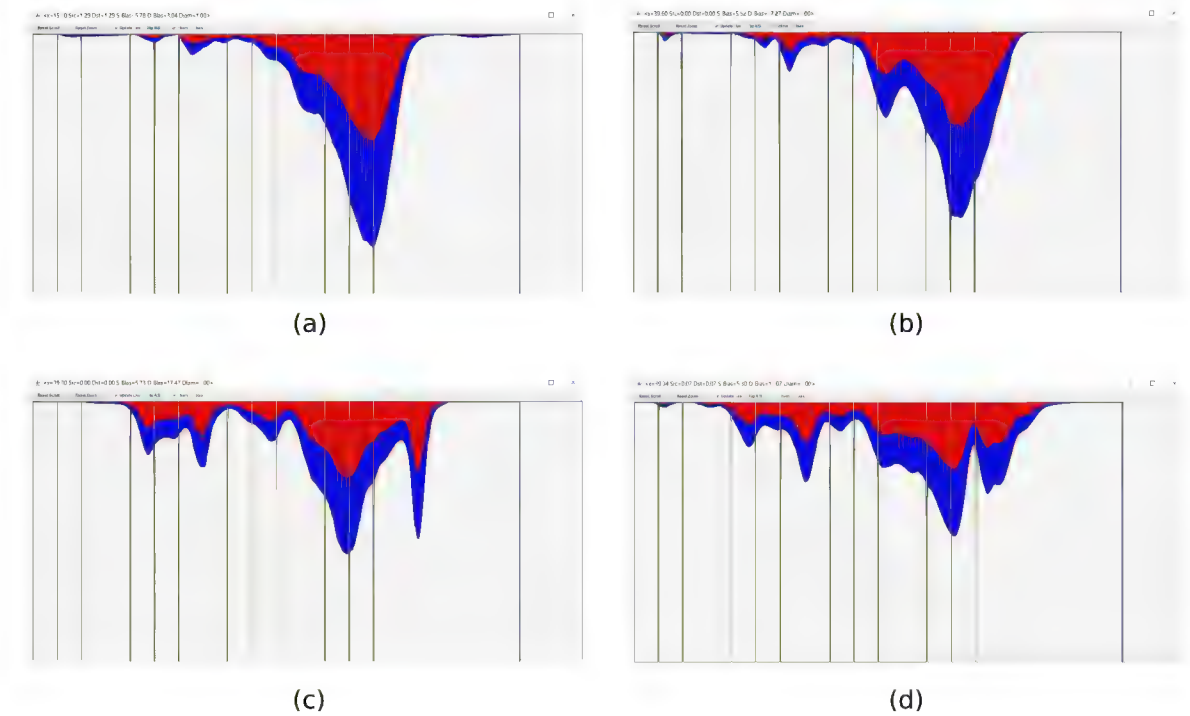
Figure 5.17: System Behavior With $\kappa = 1$. Figures (a)-(d) show 4 different timesteps. The system is changing over time, but doesn't appear to be periodic.

relative addresses.

$Diameter(x)$ specifies a relative flow metric for each spatial location. In a transfer from $MEM[SRC] \rightarrow MEM[DST]$, the final flow (based on D_κ, Δx, Δt, etc.) is multiplied by $Diameter(SRC) * Diameter(DST)$. Normally, $Diameter(x) = 1 \forall x$. Setting $Diameter < 1$ causes less of a flow rate than normal; setting $Diameter = 0$ causes no flow to occur. Note that since $Diameter(SRC)$ and $Diameter(DST)$ are multiplied, setting *either* to 0 stops any flow between $MEM[SRC]$ and $MEM[DST]$. This is useful, for example, in specifying input regions, whose chemical levels are set by external sensors. Such regions can still affect other regions of the system, but the chemical levels in those regions themselves do not change under transfer requests from inside the system. This will be discussed further in Part II.

Note that a negative diameter could theoretically be used to reverse the direction of chemical flow: this has not been explored so far.

5.8 Exercises

Exercise 5.1 Sketch a distribution of SRC and DST chemicals to transfer all chemicals from $[0,4)$ to $[20,24)$, assuming $\kappa = 0$. ∎

(a)   (b)
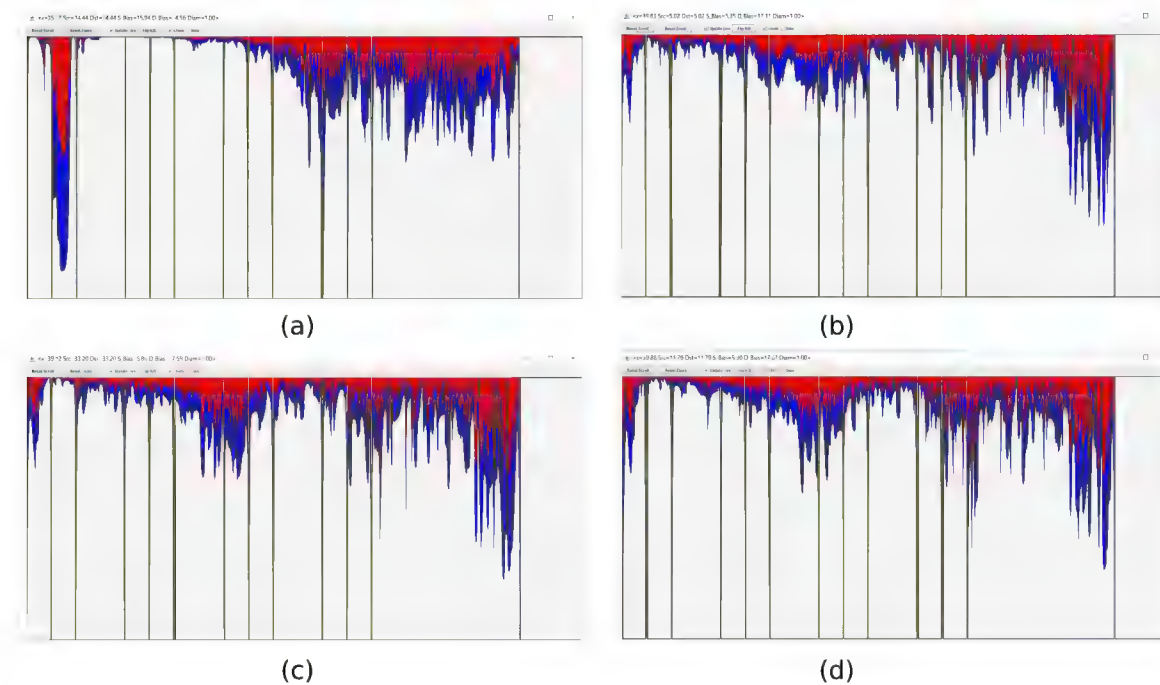
(c)   (d)

Figure 5.18: System Behavior With $\kappa = 0$. (a)-(d) show 4 snapshots of the system's chemical balance. Without the effects of karma, the system is running essentially open-loop, with each instruction modifying the system but acting independently from all other instructions.

Exercise 5.2 Repeat the above, but with $\kappa = 4$. How does the system differ from the case where $\kappa = 0$? What assumptions do you need to make?   ■

Exercise 5.3 What would be the general effect on the system if $\kappa < 0$?   ■

Exercise 5.4 What is the general effect of setting $SRC = DST$?   ■

Exercise 5.5 What is the effect of setting $BIAS_{SRC} = BIAS_{DST} = C$ where C is a constant? What if instead of being constant, C is the location where the biases are being set?   ■

Part Two - Experiments

6. Overview, Links to Software

With any proposed architecture, a natural question is: "what can one do with it?" While EEXIST was developed as an extension of more traditional computing systems, it's not immediately clear how to do anything "useful" with this system. Constructing individual transfers is simple (provided karma is ignored), but since all transfers occur simultaneously, orchestrating a sequence of actions seems difficult. While some work has been done in setting up rudimentary behaviors (nominal cyclic changes, for example), general schemes for configuring EEXIST for specific behaviors has been a challenge.

One of the early lessons was that the balance of chemicals is only part of the story. In trying to configure a system that, for example, can perform logic operations, finding a particular configuration of chemicals that leads to the desired behavior is difficult. This led to the introduction of bias and diameter, described at the end of Part I.

At that point, rather than trying to explicitly design configurations for certain behaviors, an evolutionary approach was adopted: using genetic algorithms to *discover* configurations,

rather than engineering those configurations. This approach has been used for all the work presented in the experiments in Part II.

It is important to remember that at this point, the main goal of this work is *to explore and understand what EEXIST is capable of*. Even though an evolutionary approach has been employed, this is not fundamentally research about evolvable systems or GAs. The goal is to see what behaviors EEXIST can exhibit.

6.1 Software Setup

6.1.1 Links

While this book is intended to primarily stand alone, there are references to software (mainly on GITLab) and web pages throughout the discussion of experiments. These can be clicked-on directly in the PDF version of this text. For the printed copy, the reference numbers (e.g. [1]) refer to links on http://book.songlinesystems.com (which means you don't need to copy long URLs from the text).

All this code is built on top of an API [20] which provides access to a full EEXIST simulator, including graphics displays of system activity. There is standard JavaDoc available for this API here [21]. More generally, the GIT repository for the API is available here [22]

A more-general webpage that includes links to videos discussing the EEXIST API is available at http://songlinesystems.com/eexist-api.html [23].

https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI [24] is the root of the GIT repository, containing not only the API but also the code and data for all experiments described in this text.

6.1.2 General Code Organization

Within each directory of the GIT repository (e.g. EA2), there are a number of files:

- README which often (but not always) contains helpful information about the contents of the directory;
- raw.txt and variations thereof. These usually contain genome information for individuals in evolving populations. The file is human-readable, but not easily (it would read better as a .CSV file). raw.txt is often tagged with additional information inside the filename itself (e.g. raw.xor.5.22.txt is the raw.txt from evolving an XOR gate, and individual 22 generation 5 is considered noteworthy for some reason);
- various human-readable files containing output from past runs, sometimes following manual processing;
- various scripts for monitoring output as it's being generated (especially for the tic tac toe work);
- classes Gene and Genome which handle the basic genetic makeup of individuals;

The Java code is built on top of the EEXIST API, and uses the Gene and Genome classes. Beyond that, the code is problem-specific, but there is a common structure to the code, comprised of the following pieces:

- a pair of scripts ("c" and "r") for compiling and running Java code, respectively;
- a set of code for running the evolutionary part of the system, i.e. for developing a population that performs well according to some criteria. This consists of the following code:
  - Main.java which is the main class for the evolution process; and
  - Core.java which contains the particular code for the experiments related to this directory.
- a set of code for *analyzing* the evolved population. This code generally reads from a raw.txt-type file to load saved genetic information into a test EEXIST system, and allows the user to interact with that EEXIST system in various ways. Typical pieces include:
  - Analyze.java which is the main class (analogous to Main.java)
  - AnalyzeCore.java which contains the particular code for analyzing the results of this set of experiments (analogous to Core.java); and
  - AnalyzeControl.java which contains code related to user-interactions with the cloned EEXIST system.
- a set of .jar files if the experiments require interaction with a server (e.g. LL.jar for simulating the lunar lander game, and VEco.jar for simulating the virtual ecosystem).

6.2 Genetic Setup

Early attempts at evolving EEXIST systems didn't work very well. Part of this seemed to be related to the "special nature" of certain areas of the memory. As chemicals are drained from tubes, the instructions referenced by such tubes refer to smaller and smaller SRC and DST addresses; whereas upper addresses (e.g. close to 40) are rarely references unless a tube is full. The larger issue may have been that chemicals were serving two purpose, being used both for inputs/outputs and for somehow coding the genetic signature of an individual. It is in light of this latter consideration that further parameters were introduced into an individual's genome, in order to foster a genetic signature independent of SRC/DST chemical levels in the system.

The basic genetic setup is to partition EEXIST's address space (currently $[0, 40]$) into a set of intervals (called "genes") , and within each gene, have some sort of coded variation of chemical levels (SRC and DST), biases (again SRC and DST) and diameters. The original vision for this allowed a number of different codings: fixed levels, linearly-changing levels, levels described by sinusoidal functions, and so on. The first implementation of a gene was a simplified version of this, consisting of the following:

- a specification of a single variable: either chemical amounts, bias amounts or diameter;
- an initial value for the chosen parameter at the beginning of the gene's region; and
- the slope of the linear change in the parameter across the gene.

Provisions were made to allow genes to have a variable size, and to allow for a small amount of random variation in each of these parameters.

In practice though, only the bias amounts were modulated as part of a system's genetic

signature. Additionally, all genes are the same length (4), and no variation from the specified linear change was allowed. Thus, each gene is comprised of a pair of *bias gradients* , and the entire genome is a set of 10 such equally-sized bias gradients.

Genes can be initialized with random values, or can be cloned from existing genes.

Mating occurs by simple averaging of a gradient's initial value and slope from each parent. Mutation occurs parameter-by-parameter: a random value is generated, and if it is less than the given mutation rate (e.g. 10%), then the parameter is scaled by a random amount between 0.5 and 1.5.

Most GA experiments involve an initial randomly-generated population of individuals . Each individual is assessed on some set of tasks, and scored based on its performance. The top *n* (typically 10) individuals are retained verbatim in the next generation, while all other individuals are removed. The survivors are randomly mated pairwise (with a possibility of mutation) to create the next generation .

6.3   Exercises

Exercise 6.1 Use API calls to set up an EEXIST system that is configured to do a single transfer. Run the code and observe the behavior in the graphical displays. ∎

Exercise 6.2 Use the API calls to configure an EEXIST system with a random initial configuration. Introduce more chemicals into the system and observe how the chemical balance changes over time. What sorts of behaviors can you produce? ∎

Exercise 6.3 Change the karma in the system while running the above experiments. ∎

Exercise 6.4 Look at the code for Genome.java and Gene.java. Where is mating performed? How would you change the code to select genes from one parent or the other? How is mutation introduced? What controls the number of genes? ∎

Exercise 6.5 How can you change this code to use something other than bias gradients as the genetic signature of an individual? ∎

7. Experiments in Digital Logic
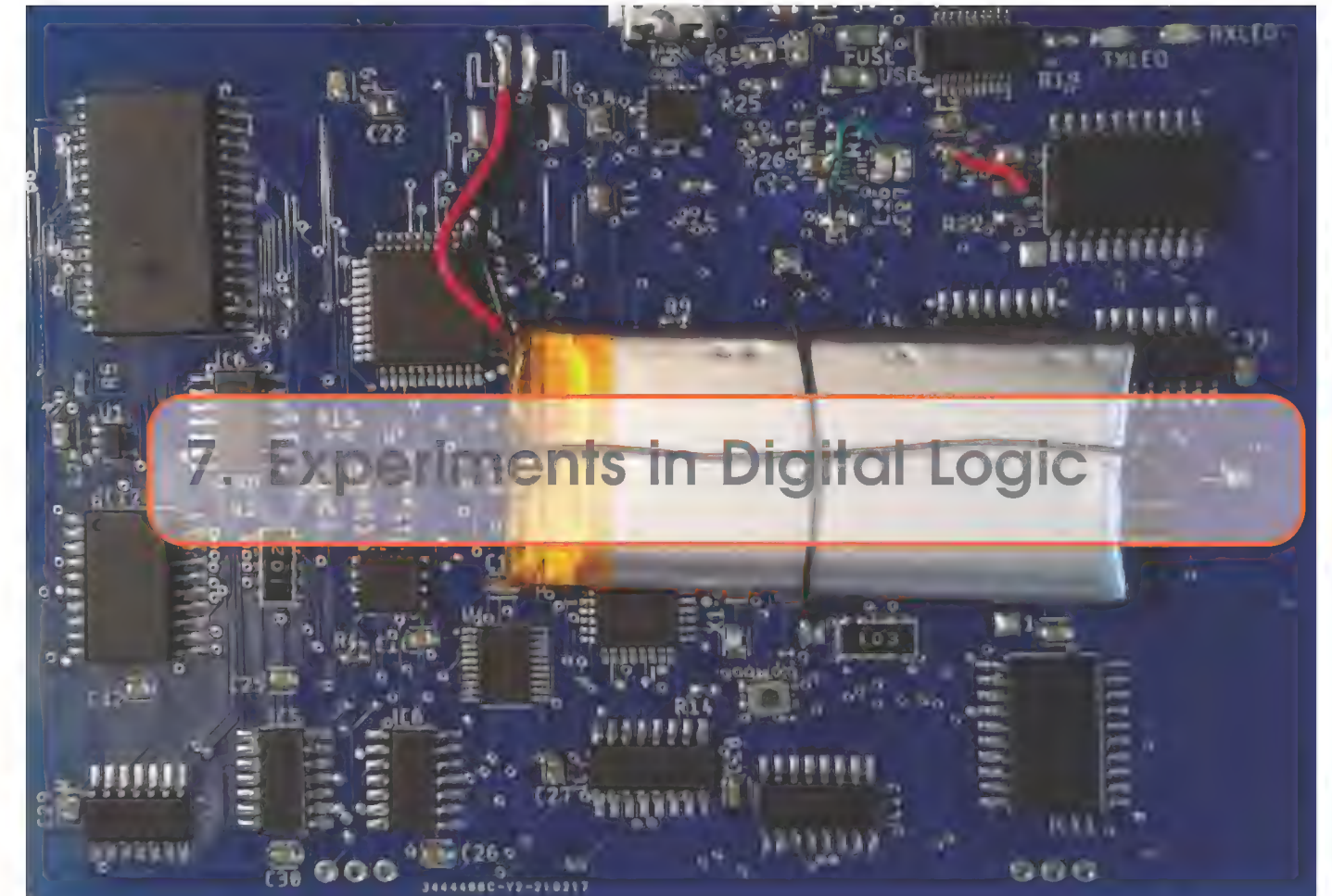
7.1   Basic Setup

The basic goal in this set of experiments was to develop EEXIST configurations that perform digital logic functions. The general setup for these experiments was roughly the same:

- space extends from coordinates 0.0 to 40.0, with $\Delta x = .0625$;
- a maximum value of 40 (for each of SRC and DST) is imposed throughout the memory;
- individuals are distinguished by their genome, which consists of 10 regions of *bias gradients*, evenly spaced between 0 and 40;
- an initial population of 250 individuals is generated with random bias gradients;
- as each individual is loaded into EEXIST (one at a time), its input-to-output behavior is monitored, and compared to the desired function;
- inputs are fed in the regions $[0,4]$, $[8,12]$ and $[16,20]$;
- an input of 1 is coded as a value of $SRC = DST = 20$; an input of 0 is coded as a value of $SRC = DST = 5$;

- the region $[24, 28]$ is considered the output region. Chemical levels at each location in this region are translated to a logic value: a chemical level of $SRC + DST \geq 15$ is considered a logic 1; $SRC + DST < 10$ is considered logic 0; anything else is considered invalid;
- an individual's score is incremented for each location on the output region (stepped by $\Delta x = 0.0625$) with the correct logic level;
- all other locations are initially devoid of chemicals;
- after injecting input chemicals, the system is stepped for 50 steps;
- the system is then stepped an additional 25 steps, during which the output is monitored;
- for each address within the output region, the individual's score (initialized to 0) is incremented if the output has the correct value. Since the output region's width is 4 and $\Delta x = 0.625$, there are 64 addresses in the output region, yielding a maximum score increment of 64 per timestep. Across 25 timesteps, this gives a maximum increment of 1600 per test;
- for a 3-input system, all 8 possible input combinations are tested; this gives a maximum total score of 12,800.
- after all individuals have been assessed, the top 10 are selected as survivors;
- pair of randomly-selected survivors are mated by averaging the start and gradient of each bias region, until the new population is 250;
- a 5% mutation rate is applied to mating.

During the evolutionary process, the genome of each individual is written to a raw output file (raw.txt). This allows individuals to be reconstructed for later testing.

The main code for the logic gate tests can be found on https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA2 [4].

7.2 3-Input Exclusive Or Gate

The first target circuit was a three-input XOR gate. This is, in some ways, an easier target circuit than an AND, OR, NAND or NOR gate. An AND gate, for example, produces a 0 in 87.5% of all test cases: so a NOP system (one that always outputs 0) would still score 11200 out of 12800. Thus, a NOP circuit (one which never transfers any chemicals into the output zone) might out-perform a promising but not-yet-fully-evolved AND gate.

Surprisingly, EEXIST evolved a perfect XOR gate in only 5 generations! Individual 22 in generation 5 scored a perfect 12800/12800 across all 8 possible input combinations. Figure 7.1 shows the bias gradients for this individual. The triples of vertical yellow lines show the input and output regions. SRC bias is in red, DST bias is in blue. For reference, the titlebar shows the SRC and DST bias values at the point indicated by the cursor arrow.

7.3 Nand Gate

The next evolve target was a 3-input NAND gate. A perfect system emerged during the 9th generation (individual 19). The raw file (raw.nand.10.0.txt) contains the genome for
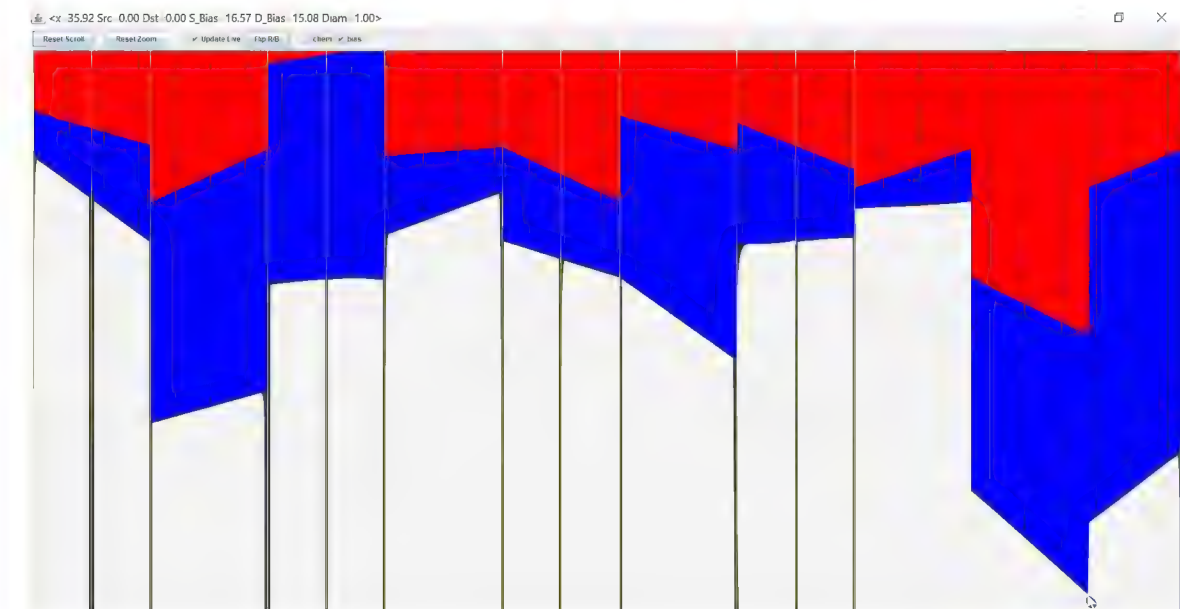
Figure 7.1: Bias Gradient For a Perfect XOR Gate. Source bias is colored red; Destination bias is colored blue. For reference, the cursor shows a point where $BiasSRC = 16.57$ and $BiasDST = 15.08$.

each individual. Figure 7.2 shows the bias gradients for a perfect individual.
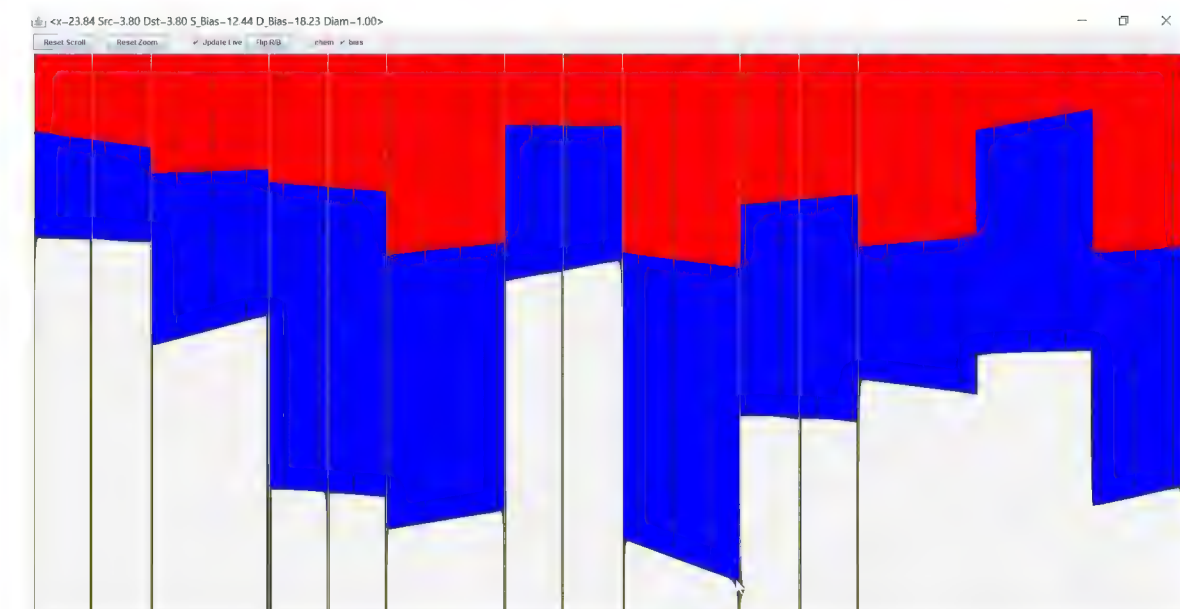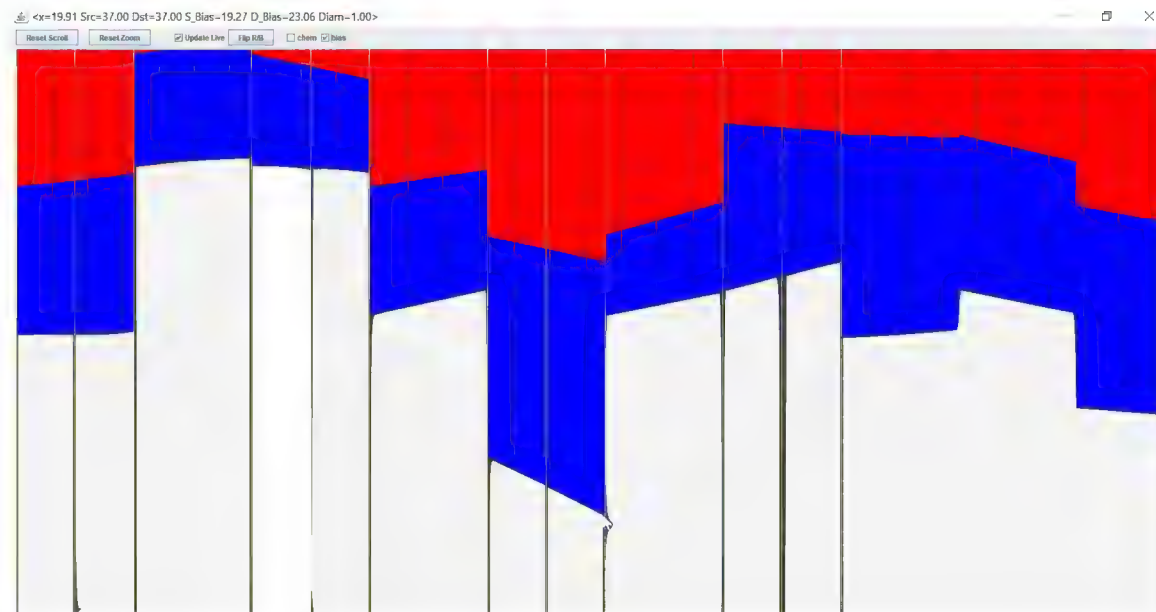

Figure 7.2: Bias Gradient For a Perfect Nand Gate. Source bias is colored red; Destination bias is colored blue. Values at the cursor are $Bias_{SRC} = 12.44$ and $Bias_{DST} = 18.23$.

7.4   Nor Gate

The next target circuit was a 3-input NOR gate. This was expected to be the most challenging of the three logic gates, because a NOR gate almost always outputs 0, so a circuit that never transfers chemicals to the output region would score 87.5%. This means an actual NOR circuit will need to score higher than this to compete with NOPs. This is significantly more challenging than a NAND gate, where a NOP circuit will score 12.5%.

A perfect system emerged during the 56th generation (individual 212). This took 5 times longer to evolve than a NAND gate, and 10 times longer to evolve than an XOR gate. The raw file (raw.nor.56.212.txt) contains the genome for each individual. Figure 7.3 shows the bias gradients for a perfect individual.

Figure 7.3: Bias Gradient For a Perfect Nor Gate. Source bias is colored red; Destination bias is colored blue. Values at the cursor are $Bias_{SRC} = 19.27$ and $Bias_{DST} = 23.06$.

7.5   Frequency Discrimination

Given that EEXIST seems to be configurable to perform logic operations, the next task was to see if it could respond to inputs over time. This test took the form of a frequency discriminator (possibly inspired by Adrian Thompson's work [6]). The main code for the frequency discriminator tests can be found on https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA3 [5]. raw1000 contains the history of individuals' genomes for this experiment.

The idea was to define an input region, and toggle the input at one of two frequencies, hoping to generate an output of 1 or 0 based on the input frequency. The test setup was as follows:

- $\kappa = 5$;
- Two input frequencies are considered:
  - high frequency changes every 4 timesteps
  - low frequency changes every 8 timesteps
- The input region is $[0,4]$: logic 1 is $SRC = DST = 20$; logic 0 is $SRC = DST = 5$;
- The output region is $[24,28]$: chemical levels at each location in the output region (stepped by $\Delta x = .0625$) are translated to logic levels: $SRC + DST > 15$ is interpreted as logic 1; $SRC + DST < 5$ as logic 0; anything else is an undefined logic level;
- an individual's score is incremented for each output that is at the correct logic level. This means a maximum possible increment of 64 per timestep;
- The input is supplied for an initial 64 timesteps;
- for the next 1000 timesteps, the output is analyzed;
- The output goal is 0 for high-frequency input, 1 for low-frequency input;
- a perfect score is 64×1000 steps $\times 2$ tests $= 128000$;

Evolution proceeded slowly, since each individual requires 1064 timesteps. Nonetheless, after 8 generations, a near-perfect individual emerged (individual 37) with a score of 127009/128000. Despite the imperfect score, all the errors were in the first 19 timesteps: meaning the last 981 timesteps were perfect across the entire output region (and actually the last 1000 timesteps were all perfect). While further evolution might have resulted in a perfect score under the original criteria, the original requirements (initialization of 64 timesteps, followed by a test period of 1000 timesteps) were essentially arbitrary, and thus the obtained result (initialization of 83 timesteps followed by a test period of 1000 timesteps) was deemed "good enough."

Figure 7.4 shows a display of the test results. In this display, time is drawn vertically (top-to-bottom, then wrapping back to the top), and space horizontally. SRC/DST amounts are color-coded (SRC is red, DST is blue). The intensity of each color reflects the amount of chemicals at that position in space and time. Since $SRC = DST$ throughout, the only color is different intensities of purple.

The vertical yellow lines delimit the input and output regions. The input region (on the left) shows the alternation between high and low chemical levels: this reflects the input signal, which has a high frequency in the top of the display, and a low frequency in the bottom. The output region (marked by two yellow vertical lines near the right of the display) shows the corresponding output:

- In the top of the display, the output is low. While faint traces of the input pattern are visible, the chemical amounts translate to a clean logic 0 throughout the test.
- When the input frequency switches to low (approximately halfway down the display), the output first goes low (dark), but then raises to a high level (bright purple) for the remainder of the time to the bottom of the display.

Figure 7.5 shows the bias gradients associated with this individual.

Figure 7.4: Space (x) and Time (y) Display of Frequency Detection Test. Time run from top to bottom (and re-wraps to the top). Intensity relates to chemical amounts. The input region is on the left, and reflects the periodic input at a high (top half) or low (bottom half) frequency. Output appears towards the right (between the two yellow lines) and shows the desired output of 0 for a high frequency input, and 1 for low frequency.

7.6   Frequency Generation

The final experiment in this set if digital logic-related behaviors was to evolve a circuit that generates an oscillating output. The setup was similar to other experiments, but with a few subtle differences:

- The input region was $[0,4]$ and the output region was $[24,28]$;
- The system was stimulated for 16 timesteps by setting the $SRC = DST = 20$ throughout the input region;
- For the next 1000 steps, the output region was monitored. In this case, the *average* chemical level ($SRC + DST$) was calculated across the region, and an average of 15 or more was considered a logic 1, otherwise the output was considered logic 0;
- An individual's score was incremented any time the output changed. Across 1000 timesteps, the maximum possible score would thus be 1000.

The code and files for this experiment can be found on https://gitlab.com/nickmacias/

Figure 7.5: Bias Gradients for Individual 37, Generation 8. Values at the cursor are $Bias_{SRC} = 17.90$ and $Bias_{DST} = 13.66$.

ChemComp/tree/master/ChemCompAPI/EA5 [7]. The file "raw" contains the genome history for each individual in this experiment. After 4 generations, individual 90 earned a score of 224, representing on average an output change every 4 timesteps. This was considered successful enough to conclude the experiment. The output from this system is shown in listing 7.1:

Listing 7.1: System Output, Generation 4, Individual 90

```
1000000000000000000000000000111100000000000000001111
1000000000000000000000110000000001100000000100000000
0010000000100000000100000001000000001000000001100000
0001000000011000000001000000011000000001000000011000
0000100000000110000000100000001000000001000000001100
0000010000000010000000001000000011000000010000000100
0000001000000010000000010000000110000000100000000010
0000000010000000100000000100000001100000001000000001
1000000010000000110000000100000001100000001000000000
1100000000100000001000000001000000011000000010000000
0110000000100000001000000001000000011000000010000000
0011000000010000000100000000100000011000000001000000
0001100000001000000010000000010000000110000000010000
0000110000000100000001100000001000000010000000100000
0000010000000010000000110000000100000010000000010010
0000001100000000100000000110000000100000001100000001
```

```
00000000110000000010000000011000000010000000110000000
10000000011000000010000000011000000010000000011000000
01000000001100000001000000011000000010000000011000000
00100000000110000000100000001100000001000000001110000
```

Figure 7.6 shows the display-view output of this run (the display is similar to figure 7.6). As usual, the input region is at the left, and the output is between the yellow vertical lines towards the right. As can be seen, after the initial input, the input region is mostly devoid of chemicals, while the output region begins to toggle between high and low shortly thereafter.
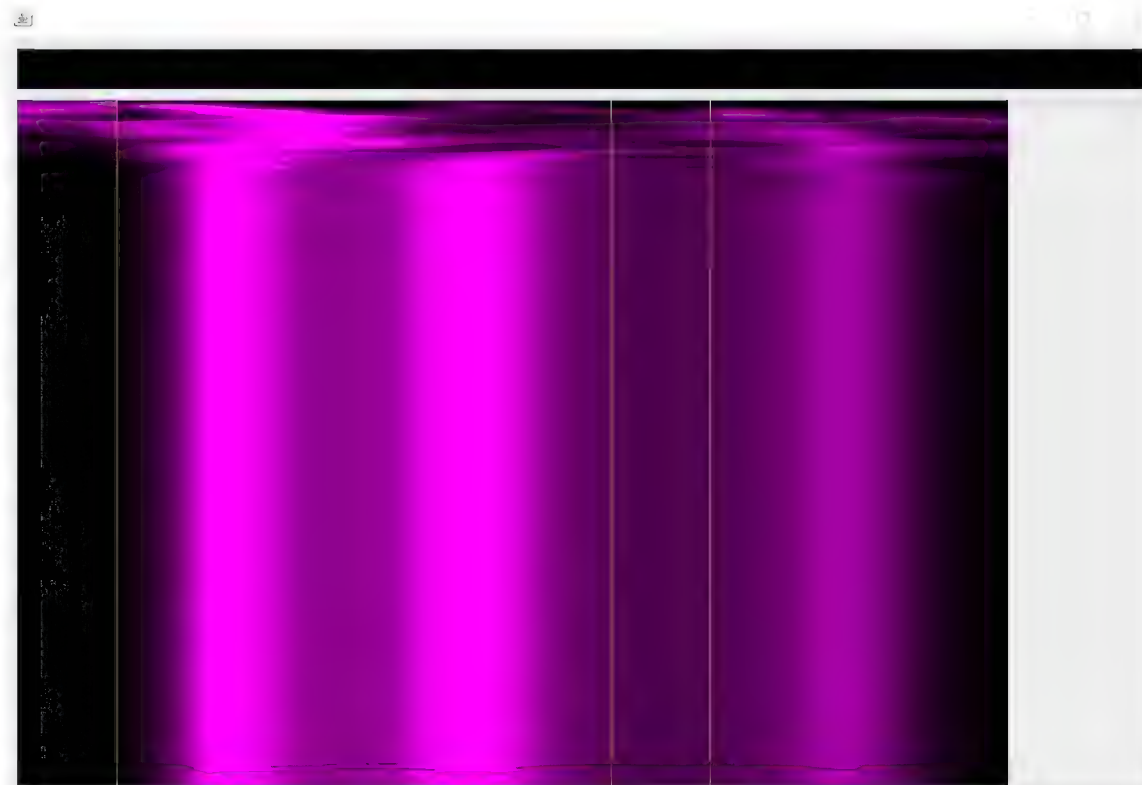
Figure 7.6: Display Window for Generation 4, Individual 90. The system is seeded with a dose of chemicals in the input region (left side). Shortly thereafter, the output region (to the right of center) begins to toggle between high and low amounts of chemicals. The output toggles 224 times during 1000 timesteps (not all timesteps are shown).

Figure 7.7 shows the bias gradients for this same individual.

7.7 Another Look at Karma

Before moving on to the next set of experiments, it's interesting to re-visit the effect of karma (κ) on the system: in particular, to see if evolution is possible without karma. Figure
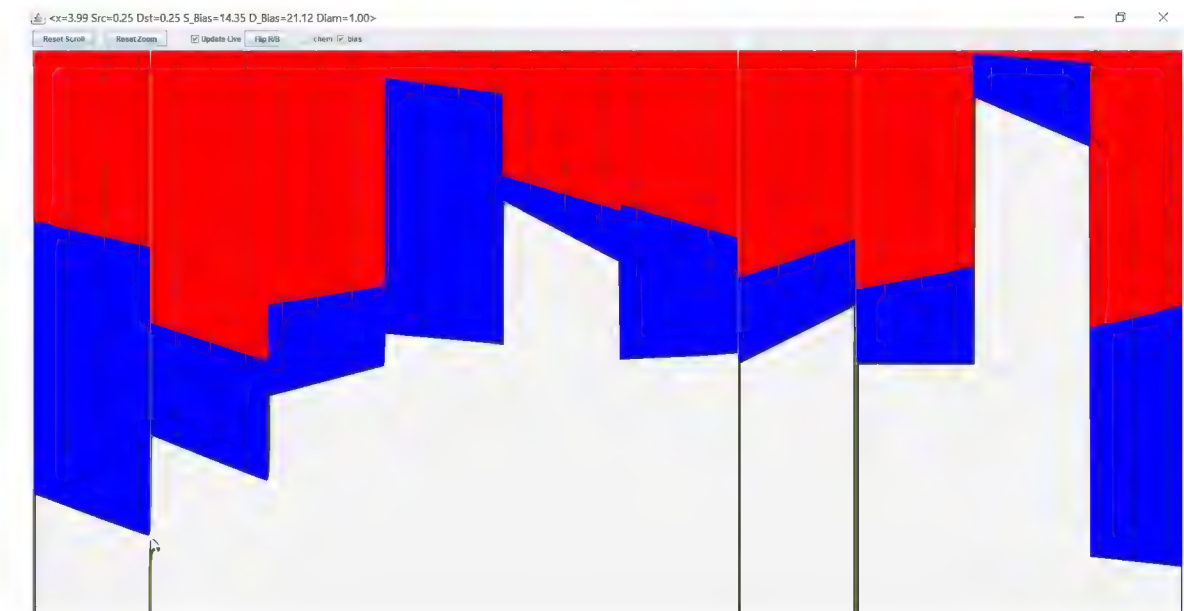
Figure 7.7: Bias Gradients for Individual 90, Generation 4. Values at the cursor are $Bias_{SRC} = 14.35$ and $Bias_{DST} = 21.12$.

7.8 shows the results of an experiment designed to explore this. The system is trying to evolve a 3-input XOR gate, using the usual general- and survivor-population sizes (250 and 10), genome structure, etc. With $\kappa = 0$, the system runs for over 1000 generations, achieving a most-fit individual with a score of 56% (i.e., it gives the correct result i 56% of the test cases). Remember that a NOP – a circuit that always outputs 0 – will achieve a score of 50%. Hence the best individual after 1000 generations is performing barely better than a circuit that always outputs 0.

During generation 1079, κ is changed to 5. Within the current population at the time of the change, one individual already performs at around 57%. Within 10 generations, the best individual is performing above 90%, and after another 21 generations, the best individual is at 99%. Thus karma seems to have a beneficial impact on evolvability.

7.8 Exercises

Exercise 7.1 Run the Analyze code on the file raw.xor.5.22.txt, and test individual 22, generation 5. Confirm that it functions as an XOR.   ∎

Exercise 7.2 Analyze individual 0, generation 1. Compare its behavior to the above. ∎

Exercise 7.3 Look at the raw file, find the entry for individual 22 generation 5, and compare the data in the file with the bias gradients shown in the simulator.   ∎
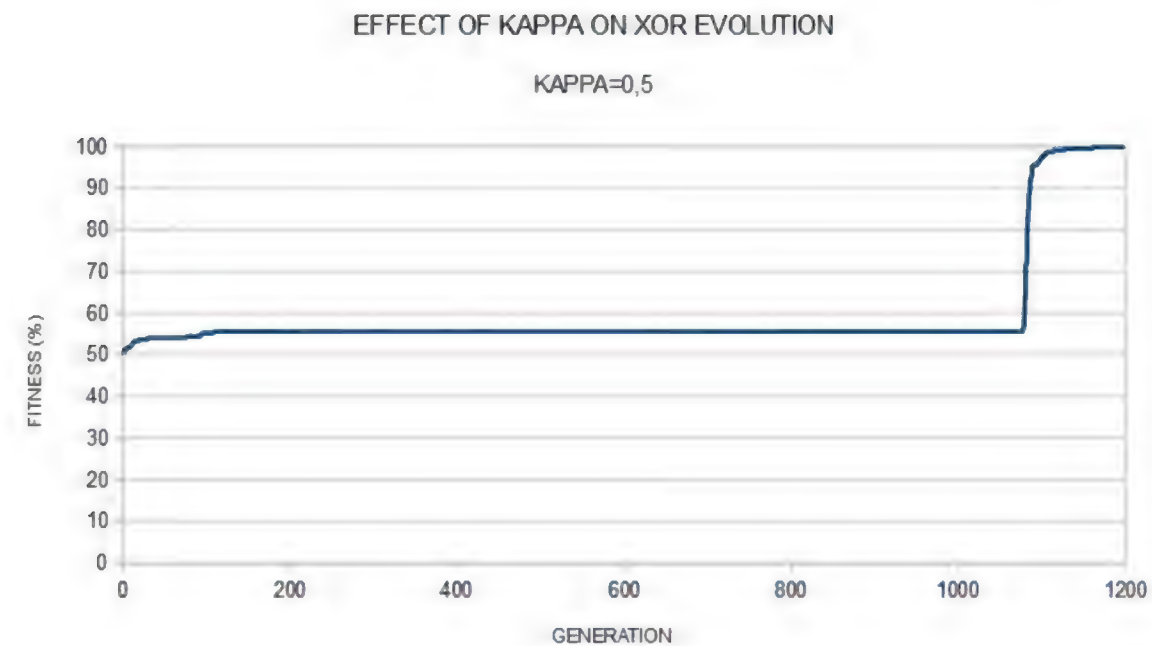
Figure 7.8: Effect of Karma on Evolution of an XOR Gate. κ is initially set to 0 and the system runs for over 1000 generations. Evolution fails to produce an XOR gate that behaves much better than a NOP. When κ is changed to 5, the system quickly evolves an almost-perfect XOR in just a few generations.

Exercise 7.4 How fragile are the bias gradients? Can you truncate to 6 digit and still get an OR? 4 digits? 2 digits? ∎

Exercise 7.5 Try evolving an XOR with truncated bias gradients. ∎

Exercise 7.6 Change the code to evolve an XNOR (look for code in Core.java around the comment that says "Test Function Here"). Run the analysis code to confirm your system performs an XNOR. ∎

Exercise 7.7 Try evolving with different values of κ. Try values such as 10, 20, 30 and 40. ∎

Exercise 7.8 Try evolving some nonstandard logic functions. ∎

Exercise 7.9 Try evolving a 4-input gate. ∎
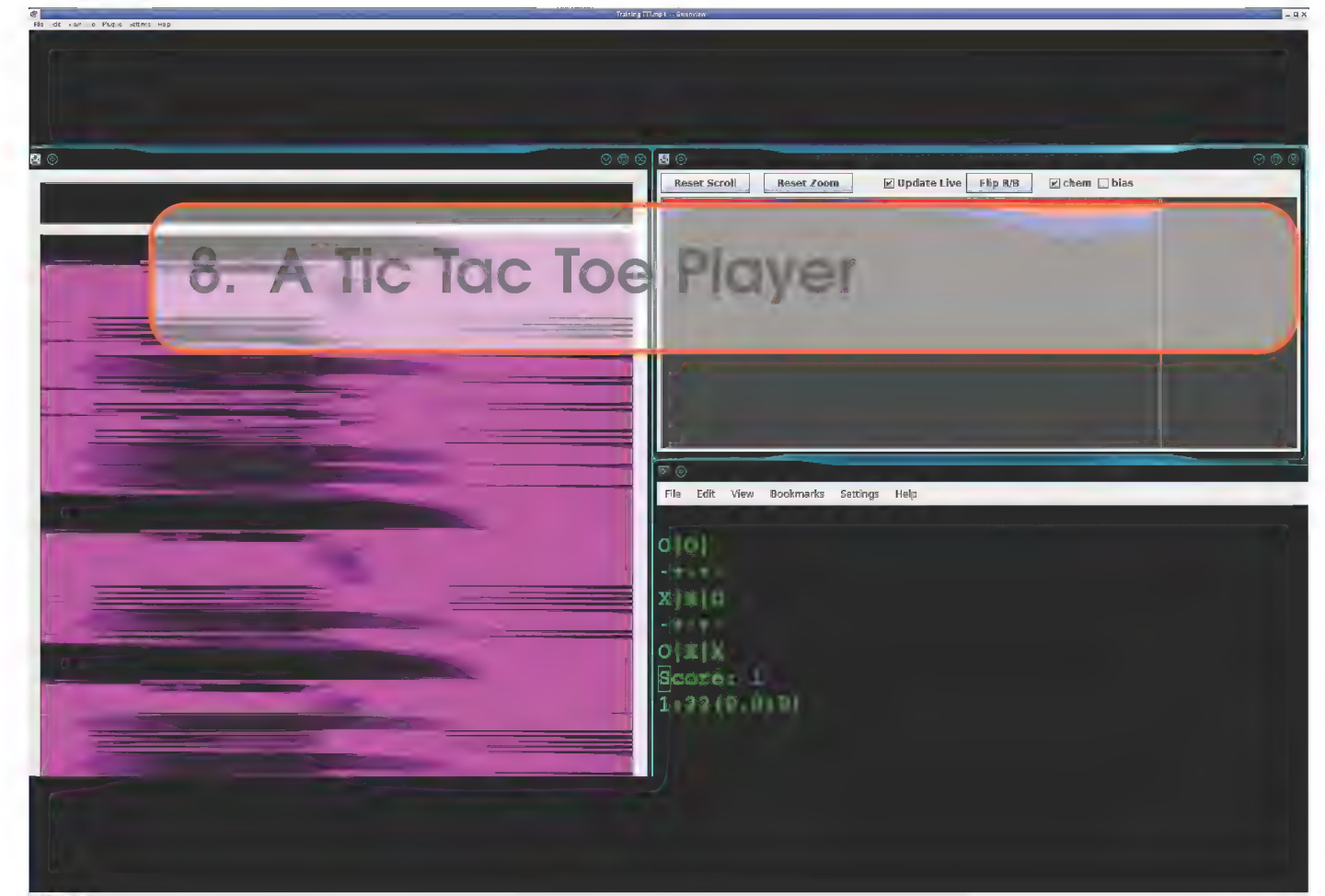
Following a series of experiments in emulating behavior related to digital logic, the next series of experiments centered around seeing if EEXIST could learn to play tic tac toe. A number of experiments were performed, with a lot of variation in setup, fitness assessment, scoring, and so on. The system quickly proved able to play a game of tic tac toe, was able to perform better than a random opponent, and was able to sometimes make good moves against an expert opponent; but it never evolved to where it could play without losing. The reasons for this will be discussed below.

The code and files for this experiment can be found on https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA6 [8].

8.1 Setup

The basic setup for all experiments was similar:
- The board was viewed as a grid of 9 squares, numbered as in 8.1;
- Each square also corresponds to a region in the EEXIST address space: region 1 is $[0,4)$; region 2 is $[4,8)$ and so on, up to region 8 which is associated with addresses $[32,36)$. These are considered output regions;
- All 9 of these regions are initially devoid of all chemicals;
- The region $[36,40)$ is used to start the system and evoke EEXIST's first move (EEXIST always goes first in these tests). To start the system, the region $[36,40)$ is filled with $SRC = DST = 15$;
- The system is then stepped, and the output regions are monitored for a move request (average of $SRC + DST \geq 10$). If a region which isn't yet marked on the board signals a move request, that is considered EEXIST's move. The square is given EEXIST's mark, and it becomes the opponent's turn;
- Once the opponent has selected a move, it is indicated to EEXIST by saturating the selected region with $SRC = DST = 15$ (the same as for the initiation in $[36,40)$);
- Play continues until someone wins, or there are no spaces left (in which case the game is a draw).

0 [0,4) a	1 [4,8) b	2 [8,12) c
3 [12,16) d	4 [16,20) e	5 [20,24) f
6 [24,28) g	7 [28,32) h	8 [32,36) i

Figure 8.1: Layout of Tic Tac Toe Board. Regions denoted $[x_0, x_1)$ are the EEXIST address range associated with that square. Indexes 0 through 8 are used internally in the code; letters "a" through "i" are used in playing an interactive game.

Note that no check is made to prevent illegal moves by the opponent.

There is a maximum number of timesteps the system will wait for EEXIST to move (typically 250 steps). If no move is detected in that time, EEXIST is considered to have forfeited.

The genetic algorithm was run with an initial population of 250 random configurations, a survivor size of 10 individuals, and a mutation rate of 5% (the same as in the digital experiments). The genome was again a set of 10 evenly-spaced bias gradients. Mating was a simple averaging of the parents' biases, point-by-point.

EEXIST was always "X" and the opponent was always "O" (this doesn't affect anything, but is useful to know if one is looking at the code).

A lot of output was generated during these evolutionary runs. Besides writing the genome of each individual (the "raw..." files), a graphical display of the board was also produced. This was often saved to a file, and read offline.

This was also the first set of experiments to be run on an external server. Since the server had no X interface, a headless version of the code was produced (these are the "...HL" variants of the directories).

8.2 Goals

It's important to remember that the overarching goal of this work is not to play tic tac toe (a simple BASIC program will do that); nor is it to study genetic algorithms or evolvability *per se*. Rather, the goal is *to learn what types of behaviors EEXIST is capable of, to better understand its capabilities and limitations*. With that in mind, trying to evolve a system that play tic tac toe seemed in interesting endeavor; and as usual, rather than trying to manually engineer EEXIST to do this, an evolvable approach was explored.

The immediate goal was, as always, to evolve individuals with higher and higher fitness measures. However, there are a number of possible ways to assess fitness. For example, the following are all slightly different goals for a Tic Tac Toe playing system:
- play better than a random opponent;
- play better than an "average" opponent;
- win more than you lose;
- never lose (but possibly draw sometimes);
- play a perfect game.

An exact understanding of the goal(s) will help define the fitness metric. However, there are tensions among the above goals:
- training against random opponents may lead to a system that scores well, i.e., has a great fitness measure, but fails miserably against a skilled opponent. For example, winning 90% of all games against random opponents might still allow for 100
- training only for wins ignores the fact that not all games are winnable. By moving first, one can be guaranteed to never lose, but a skilled opponent can always force a DRAW. Therefore, if training against a perfect opponent, one would never see a single win. This intuitively goes against the idea of ranking WIN higher than DRAW.
- If the system is trained against every possible game, and WIN is ranked higher than DRAW, it's possible a system that is defeatable by a perfect opponent might not score as well overall as one that wins many games against imperfect opponents.

In many cases, the system seemed to gravitate towards local maxima, tending to find excellent performers among groups of sub-optimal individuals.

Scoring was based on the outcome of each game, with the following possibilities:
- winning;
- losing;

- ending in a draw; and
- forfeiting (no move made),

along with the following intermediate metrics:

- failing to take a winning move when presented with the possibility; and
- failing to block an opponent's winning move when presented with the possibility.

Different scoring metrics were used, based on the above measures. For example:

- $WIN = +5$
- $DRAW = +2$
- $LOSE = -1$
- $FORFEIT = -\infty$

Forfeiting was always treated as immediate dismissal of the individual (they were given a final score that was negative). This was largely due to practical concerns: whereas many moves are made after only a few timesteps, forfeiting means waiting (e.g.) 250 timesteps before giving up on EEXIST's making a move. This simply slowed down the evolutionary process too much, hence such individuals were quickly removed from the population.

Evolution depends on the nature of the population of opponents:

1. in a totally random population, opponents make random (legal) moves;
2. in a population of smart individuals, opponents make winning moves if possible, but otherwise make random moves;
3. in a population of smarter individuals, opponents block EEXIST's winning moves if possible, but otherwise make random moves;
4. in a population of perfect individuals, opponents always play a perfect game (and thus never lose);
5. in an exhaustive population, all possible games are played.

Of course, combinations of these (such as 2 and 3) are possible.

8.3   Results

Directories from these runs are available on GITLab, generally in subdirectories underneath https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI In general, there are two sets of code:

- Main.java and Core.java are the heart of the *evolutionary* system (along with support code: Game, Gene and Genome);
- Analyze and AnalyzeCore are the counterparts to Main and Core, but are used for *analysis* of individuals (along with AnalyzeControl, plus Game, Gene and Genome).

The evolutionary code displays gameplay while it runs, but also writes a raw file of individual's genomes. This file can be loaded into the analysis code, which allows the user to play against an EEXIST-controlled opponent. In this mode, the user specifies moves by naming the square (using the letters shown in figure 8.1).

A number of different tests were run; only two are described below.

8.3.1   EA7HL

The EA7HL directory contains the code and results for the following test setup:

- 50 games per individual
- 250 timesteps maximum wait time for a move from EEXIST
- Scoring: Win=2, Draw=1, Loss=0, Bad Move=-1 (where a "Bad Move" means failing to take a winning move, or block an opponent from winning their next turn)
- Final score is the simple sum of each game's score

With 50 games, and a maximum score per game of 2 (for winning), the best possible score for an individual is 100. Against a well-trained opponent, the best possible score is 50 (50 draws). The best individual in this set of experiments (file "rawgood" generation 28 individual 0) scored 61. Figure 8.2 shows the bias gradients for this individual.

This individual always moves in the middle square ("e") first. Based on the opponent's first move, the outcomes are as follows:
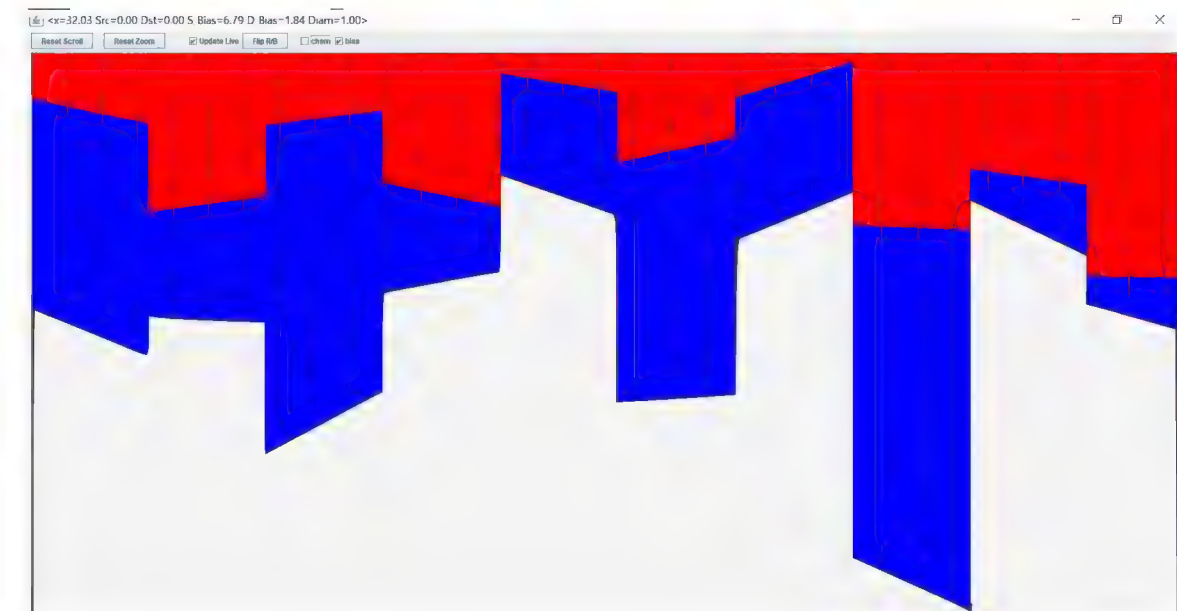
Figure 8.2: Bias Gradients For EA7HL, Individual 0, Generation 28. Values at the cursor are $Bias_{SRC} = 6.79$ and $Bias_{DST} = 1.84$.

- a: game is a draw (but if the user doesn't block EEXIST, they can actually win)
- b: EEXIST finds 2 ways to win, and eventually wins, *as long as the user tries to block*
- c: game is a draw (but if the user doesn't block EEXIST, they can win)
- d: user ends up with a choice of 2 moves: one leads to a win for the user, the other leads to a draw
- f: game is a draw
- g: game is a draw

- h: EEXIST finds 2 ways to win: blocking one leads to EEXIST winning; blocking the other leads to a draw
- i: game is a draw

This set of experiments was the first clue to the system's sensitivity to training data. For example, when the user's first move is "a," EEXIST eventually is one move from winning: but if the user doesn't block that move, EEXIST may fail to take that move its next turn.

Figures 8.3 - 8.5 show three possible outcomes for this initial user move. In 8.3, everyone plays as expected, and the game is a draw. This is the expected behavior. In 8.4, the user fails to block EEXIST from taking a winning move, and in fact EEXIST wins on that next move. In 8.5, the user blocks EEXIST the first time, but fails to block EEXIST's second possible winning move. In this case, EEXIST fails to take that winning move, allowing the user to win on their next turn.

This illustrates some of the challenges of training the system: if the training opponent always blocks, then EEXIST may be tripped up by an opponent who doesn't block; but if the training opponent misses some blocks, EEXIST may score unreasonably high due to these overly-simple games.

8.3.2   EA8HL

The "EA8HL" directory contains code and output for a series of experiments where EEXIST plays *all possible games*. Specifically, for each move EEXIST makes, every legal opponent move will be considered. In theory, since there are 9 squares, and EEXIST moves first, there are 8 possible first moves by the opponent; after EEXIST's 2nd move, there are 6 open squares for the opponent to choose from; and so on. Thus there are a *maximum* of $8 \times 6 \times 4 \times 2 = 384$ possible games. In practice, the number is fewer than this, since some of these games may end before the opponent has made 4 moves.

An array ("moves[]") is used to record the current set of moves an opponent will make, and this is incremented after each game. If a game ends before a total of 9 moves, the game tree is pruned to remove non-viable options.

Scoring was based on the following scale:

- Win=+2
- Draw=1
- Lose=-2
- forfeit=-1 (immediate end of testing, with a final score of 0 for the individual)
- badmove=-1 (immediate end of testing, with a final score of 0 for the individual); a "badmove" is failure to block or failure to take a winning move

(In fact, various other scoring criteria were explored, mostly non-scientifically. The above were the final values used).

Given such a potentially large number of games per run, evolution proceeded slowly in this set of experiments. As such, only 19 generations were developed. See the file "raw18," generation 19, individual 0 for the top performer. Figure 8.6 shows the bias gradients for this individual. Using the reference letters from figure 8.1, the only way the user can win is by moving in squares "f," "a," "b" and then "c"; any other set of moves leads to a draw

```
    a|b|c                           O|b|c
    -+-+-                           -+-+-
    d|X|f                           d|X|f
    -+-+-                           -+-+-
    g|h|i                           g|h|i

1. EEXIST starts in the middle   2. User moves in upper-left
---------------------------------------------------------------
    O|b|c                           O|b|c
    -+-+-                           -+-+-
    d|X|X                           O|X|X
    -+-+-                           -+-+-
    g|h|i                           g|h|i

3. EEXIST moves in f and is      4. User blocks EEXIST in d
   ready to win
---------------------------------------------------------------
    O|b|c                           O|b|O
    -+-+-                           -+-+-
    O|X|X                           O|X|X
    -+-+-                           -+-+-
    X|h|i                           X|h|i

5. EEXIST blocks user in g       6. User blocks in c
---------------------------------------------------------------
    O|X|O                           O|X|O
    -+-+-                           -+-+-
    O|X|X                           O|X|X
    -+-+-                           -+-+-
    X|h|i                           X|O|i

7. EEXIST blocks in b            8. User blocks in h
---------------------------------------------------------------
    O|X|O
    -+-+-
    O|X|X
    -+-+-
    X|O|X

9. Game is a draw
```

Figure 8.3: Generation 28, Individual 0. If the user plays predictably (i.e. as a "smart" user would), the game will eventually be a draw.

```
    a|b|c                              O|b|c
    -+-+-                              -+-+-
    d|X|f                              d|X|f
    -+-+-                              -+-+-
    g|h|i                              g|h|i

1. EEXIST starts in the middle    2. User moves in upper-left
---------------------------------------------------------------
    O|b|c                              O|O|c
    -+-+-                              -+-+-
    d|X|X                              d|X|X
    -+-+-                              -+-+-
    g|h|i                              g|h|i

3. EEXIST moves in f and is       4. User does not block
   ready to win                      (moves in b instead of d)
---------------------------------------------------------------
    O|O|c
    -+-+-
    X|X|X
    -+-+-
    g|h|i

5. EEXIST takes the win
```

Figure 8.4: Generation 28, Individual 0. If the user fails to block when EEXIST is one move from winning, EEXIST takes the winning move and winds the game.

(assuming the user makes "smart" moves, i.e., blocks when EEXIST is one move from winning). Failing to block sometimes (but not always) allows EEXIST to win.

8.4 Lessons and Next Steps

One lesson learned from these experiments was the importance of choosing a good survival metric. In the end, there didn't seem to be a reasonable way to score an individual in a feasible amount of time. Some of the setups were allowed to run for a few weeks, with the evolution continually producing higher-scoring individuals, but the improvement was very slow, and it wasn't clear if these improvements were more than incremental. This was, in large part, a failure in the scoring system (which was somewhat biased to faster execution, i.e., not allowing individuals who forfeit to survive, when in fact their genome might have eventually contributed to a more-fit individual).

A second issue was the mapping of the board to EEXIST's spatial domain. The decision to have EEXIST's space run from 0 to 40 was made in the earliest days of the simulation,

```
    a|b|c                              O|b|c
    -+-+-                              -+-+-
    d|X|f                              d|X|f
    -+-+-                              -+-+-
    g|h|i                              g|h|i

1. EEXIST starts in the middle    2. User moves in upper-left
---------------------------------------------------------------
    O|b|c                              O|b|c
    -+-+-                              -+-+-
    d|X|X                              O|X|X
    -+-+-                              -+-+-
    g|h|i                              g|h|i

3. EEXIST moves in f and is       4. User blocks EEXIST in d
   ready to win
---------------------------------------------------------------
    O|b|c                              O|O|c
    -+-+-                              -+-+-
    O|X|X                              O|X|X
    -+-+-                              -+-+-
    X|h|i                              X|h|i

5. EEXIST blocks user in g         6. User does not block
                                      (moves in b instead of c)
---------------------------------------------------------------
    O|O|c                              O|O|O
    -+-+-                              -+-+-
    O|X|X                              O|X|X
    -+-+-                              -+-+-
    X|X|i                              X|X|i

7. EEXIST fails to take the win    8. User wins
   and moves in h instead
```

Figure 8.5: Generation 28, Individual 0. In this case, the user blocks EEXIST's first winning opportunity, but fails to block its second winning opportunity. EEXIST fails to claim that move, allowing the user to win on their next move.

and persisted throughout this work. Breaking that range into 10 equal-sized intervals and using characteristics of each interval as a genome was proposed in the first genetic work on EEXIST, and has simply never been changed. However, given 9 squares on the tic tac toe board (and an obvious mapping from board squares to regions in the genome), and a 10th region for initiation of the game, there was, in some sense, no space left over for
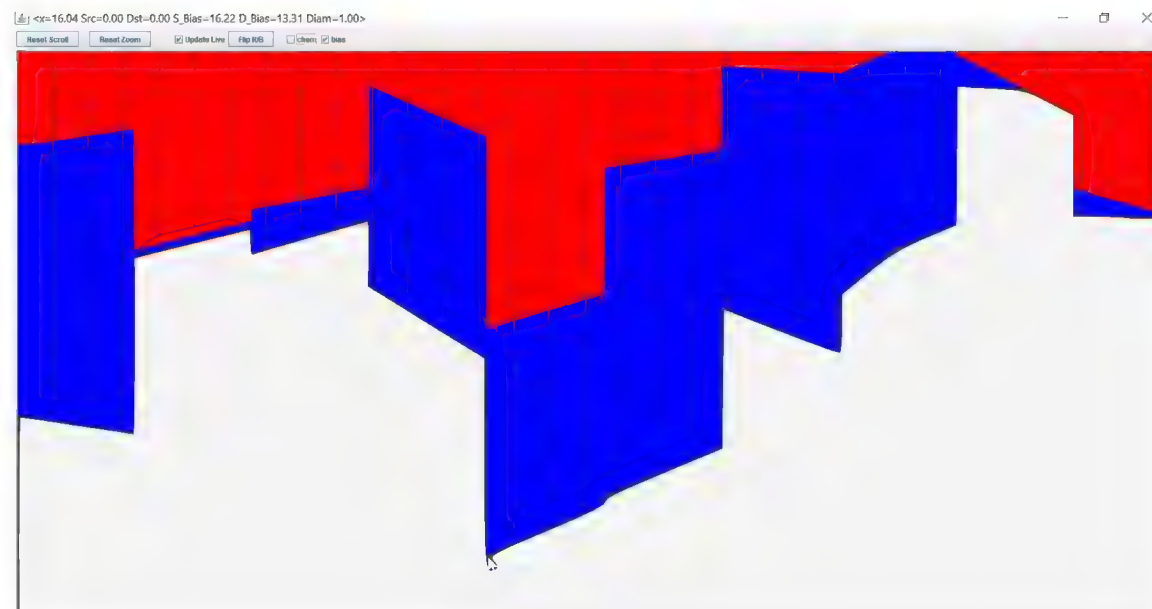
Figure 8.6: Bias Gradients For EA8HL, Individual 0, Generation 19. Values at the cursor are $Bias_{SRC} = 16.22$ and $Bias_{DST} = 13.31$.

calculations other than [36,40]. Any intermediate work being done via chemical transfer was likely signaling (or contributing directly to the eventual signaling of) a desired move. It may be that having more unallocated space would lead to better performance.

At this point though, it's worth going back and reconsidering the basic question: "What are we trying to do?"; and again, the answer is neither "Play tic tac toe," nor "Study genetic algorithms for game playing." The goal is to study EEXIST's capabilities. The fact that it will play a tic tac toe game at all is somewhat remarkable; the fact that it can lead most games to a draw, and sometimes force a win, is perhaps even more interesting. The takeaway message though is that EEXIST *does* seem able to exhibit a range of behaviors: nothing that can't be done on a von Neumann machine, but given its peculiar nature and the difficulty of manually configuring the system, the fact that there *are* configurations that do interesting things is noteworthy, and suggests further investigation may be interesting.

In continuing to pursue a genetic/evolutionary approach though, the question of a "good" survival metric needed to be addressed. Rather than ponder different weightings for different actions, or how to combine individual test scores into a composite score for an individual, a different approach was adopted: testing individuals in an environment where they might survive or perish; and using their survival as the sole metric for passing on their genes. In other words, test individuals in a system where they will either survive or perish. As long as they survive, they can mate. If they perish, their genes no longer contribute directly to figure generations of individuals.

A few years ago, I had a student named Jordan Curry, who developed a *Virtual Eco System* ("VEco") for his multi-term project. When he demonstrated this to me, it seemed

like a powerful environment in which to explore various algorithms for survival and development. In discussing this in the light of EEXIST work, we decided it might be interesting to drive each of his individual creatures with an EEXIST. The system already incorporated a semblance of mating, so this seemed like a natural framework for exploring EEXIST.

To do this required two main development though:

1. A new version of VEco, into which could be tied a population of EEXIST individuals; and
2. A way to feed inputs to EEXIST over a long period of time without saturating the system.

Item 1 simply required time and coordination. Item 2 is more of a general issue. Inputs are being modeled by injecting chemicals into a region of EEXIST's memory; but since those chemicals may be drained off to other locations (due to various transfer commands), continually restoring an input region's chemical levels to a fixed value nay result in more and more chemicals being injected into the system, leading to saturation. This may or may not be a problem: but it felt like it would lead to a weakening of a system's effects over time. This was remedied using the *diameter* option (discussed previously).

As a preamble to VEco, an intermediate task was undertaken: using EEXIST to control a simplified lunar landing game. This is the topic of the next chapter.

8.5 Exercises

**Exercise 8.1** Run the analyzer and see how well EEXIST plays. Try playing as a skilled player, a random player, or a deliberately unskilled player.

Exercise 8.2 Come up with a new grading scheme and try to evolve the system.

Exercise 8.3 Change the size of input regions and re-evolve.

Exercise 8.4 Make a 2nd EEXIST system and play EEXIST against itself. Does it get better, or plateau at some nominal skill level?

9. A Lunar Lander Controller

As an exploration of a different problem space, experiments were performed next to see how EEXIST might control a simplified lunar lander game. The idea was similar to the old 1970s Lunar Lander video game [9], but simplified in a few ways:

- the landing surface was flat;
- thrust is a simple binary control (on/off); and
- there is no horizontal control, only vertical.

The code and outputs for these experiments can be found at https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA10HL [10] and https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA11HL [11]. The system is setup as a client/server pair (another preparatory step for VEco).

The code for the landing simulator (server) itself can be found here [12] while the usual Main/Core/etc. in EA10HL and EA11HL contain the client code driven by EEXIST.

9.1 Simulation Setup

The simulator itself runs as a server: the code is in https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/LunarLander [12]. It establishes a ServerSocket on port 1213, through which the user or client code can send initial conditions, step the simulation, turn the thrust on or off, and request information about the system's status (altitude , speed and remaining fuel). The simulator also shows a graphical display of the ship's progress, and a crude visual display of landing speed (a red circle whose diameter corresponds to landing speed). The simulator is runable as a jar file ("LL.jar") in https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA10HL [10].

The simulation begins with a craft at a given initial altitude (e.g. 300), being pulled down under a constant gravitational pull (-9.8, which of course sounds reasonable on Earth, though not so much on the moon), with a given initial amount of fuel (30). Thrust is normally off, but fuel is burned at a constant rate of 1 (unit) per second while thrust is applied. The craft's initial velocity is 0. A simulated timestep of $\Delta t = .125$ is used to step the simulation.

The following equations summarize typical initial conditions:

- $a_g = -9.8$ (acceleration due to gravity)
- $a_T = 12$ (acceleration due to thrust)
- $f = 30$ (fuel)
- $s = 300$ (initial altitude)
- $v = 0$ (initial velocity)
- $\Delta t = 0.125$ (simulation timestep)

At each timestep, the system is updated as follows:

- $f = f - \Delta f \times \Delta t$ is thrust is on
- $a = a_g$ if thrust is off; $a = a_g + a_T$ is thrust is on and $f > 0$ (total acceleration)
- $v = v + a \times \Delta t$
- $s = s + v \times \Delta t$

The simulation ends in one of two cases:

- $s < 0$ In this case, the craft has hit the surface. The system's score for this run is $|1/v|$ (so a lower impact speed gives a higher score);
- $s > 1000$ In this case, the system has likely locked the thrust on, will exhaust its fuel and eventually crash into the surface. The system's score for this run is $1/s_{min}$, the reciprocal of the minimum altitude (so the closer it got to the surface before reversing velocity, the higher the score).

9.2 EEXIST Interface

The lunar lander simulation can be driven manually, via a telnet connection to the server's port. A simple command-line interface allows setting of thrust on or off, as well as interrogation of current system system (altitude, velocity and remaining fuel). The goal of this set of experiments was to have EEXIST control the lander. This means supplying current system information to EEXIST, and having it output thrust commands (ON or

OFF). The basic mechanism was similar to that used in the tic tac toe player: certain regions were defined as input and output regions, and chemical levels corresponded to input and output values.

Three input regions are defined:

- FUEL: the amount of fuel remaining.
  - Typical range: 30 to 0
  - Input address range: $[0,4)$
  - Formula: $SRC = DST = fuel \times 1.5$
- ALTITUDE: the current altitude of the lander
  - Typical range: 300 to 0
  - Input address range: $[8,12)$
  - Formula: $SRC = DST = altitude/60$
- VELOCITY: the current velocity of the lander
  - Typical range: 10 (rising) to -30 (falling)
  - Input address range: $[16,20)$
  - Formula: $SRC = DST = altitude/60$

There is also an initialization region ("GO"), defined at $[36,40)$, which is initialized with $SRC = DST = 20$ at the beginning of the experiment.

There is one output region defined: THRUST, which is a binary output, defined at $[24,28)$. $SRC + DST$ is examined at all points throughout that region, and the average value is interpreted as follows:

$$THRUST = \begin{cases} ON & \text{if } average \geq 10 \\ OFF & \text{if } average < 10 \end{cases} \qquad (9.1)$$

Simulation begins by populating the input regions, injecting the GO signal, and initially stepping EEXIST one timestep. Simulation then proceeds as follows:

1. the THRUST value is determined from the output region;
2. the corresponding thrust command is sent to the simulator;
3. the simulator is stepped forward one timestep;
4. the simulator is queried as to the current state of the system;
5. EEXIST's input regions are updated accordingly; and
6. EEXIST is stepped forward one timestep

The above steps are repeated until the craft lands, or its altitude exceeds 1000 (these conditions are reported by the simulation server).

If the craft has landed, the score is $|1/v|$ where v is the impact velocity. If the altitude exceeds 1000, then score is $0.1/alt_{min}$ where alt_{min} is the minimum altitude achieved during the run (note that at alt_{min} the velocity must have reached 0).

If multiple tests are run, the scores for all tests are multiplied to give the final score for the individual.

9.3 Sets of Experiments

Whereas in the tic tac toe system and individual was tested against hundreds of opponents, in the lunar lander game controller each individual was (in some cases) scored on a single test only. The system was given control of the craft's thrust; allowed to land the craft; and then scored on its performance.

A wide range of experiments were performed, with various modifications made, including:

- cutting thrust if the craft is close to the surface (since it often seemed to get close and then reverse direction);
- changing the mutation rate during breeding (e.g. breeding 50 individuals with no mutation; 50 individuals at a mutation rate of 5%; 125 at 10
- changing the scaling for scoring based on minimum altitude;
- changing the population size;
- randomizing the initial altitude at the beginning of each population test (but leaving it the same for all members of that population); and
- testing each individual at a variety of altitudes.

It was this last variation that produced the most interesting results, which are described in the next section.

9.4 Some Results

Among the experiments performed on this system, the most interesting are contained in the files raw8-raw11 (with corresponding output files out8-out11). All these runs were made with initial fuel=30, g=-9.8, thrust=12.0, and $\Delta t = 0.125$.

raw8 was developed using an initial altitude of 300. Individual 0 (generation 87) lands at an impact speed of -0.931 (this is considered a successful landing). However, the performance is extremely sensitive to initial velocity: at an initial altitude of 299, the impact velocity is -2.4; while at an initial altitude of 301, the impact velocity is -42.8!

Figure 9.1 shows the impact velocity for a range of initial altitudes. As can be see, while the performance is good at an initial altitude of 300, it degrades quickly away from that point. Figure 9.2 shows the bias gradients for this individual.

In file raw9, individuals were assessed at a range of altitudes, from 295 to 305 (by 1s, i.e., 11 different initial altitudes). Each run was scored, and the individual's final score (which was used for the evolutionary process) was the product of scores from each run.

Figure 9.3 shows the impact velocity for different initial altitudes. In this case, the performance seems to drop off linearly away from 300, continually degrading below 300, while actually improving towards 400. Figure 9.4 shows the bias gradients for this individual.

In raw10, each individual was assessed at 11 different initial altitudes, from 275 to 325 (by 5's). In this case (see figure 9.5), the performance remains good at values below 300, all the way down to an initial altitude of 200; while at altitudes greater than 300, the performance degrades steadily (but remains good in the training range of 275-325). Figure
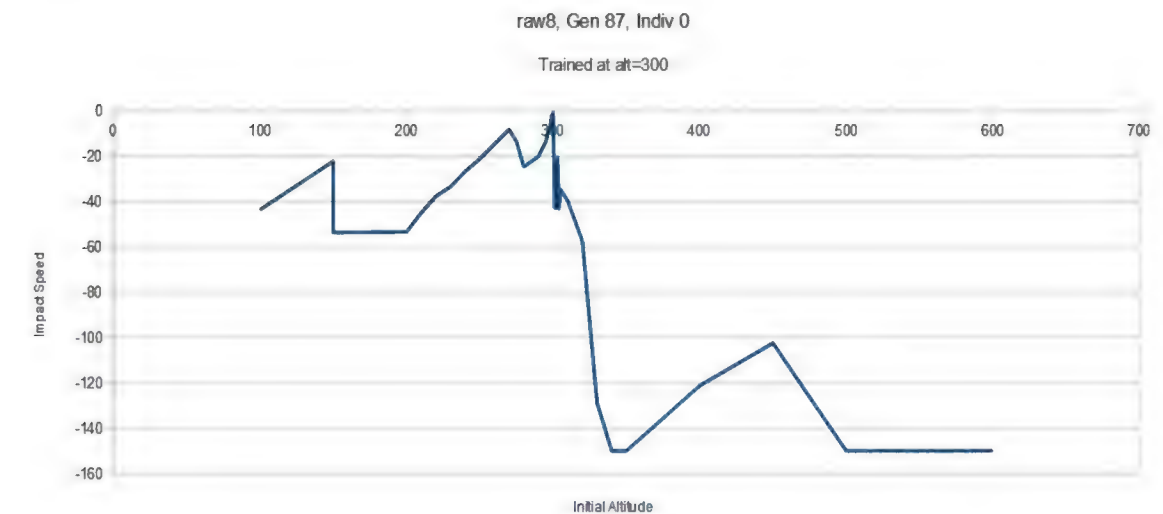
Figure 9.1: Impact Velocity vs Initial Altitude, raw8, Generation 87, Individual 0.
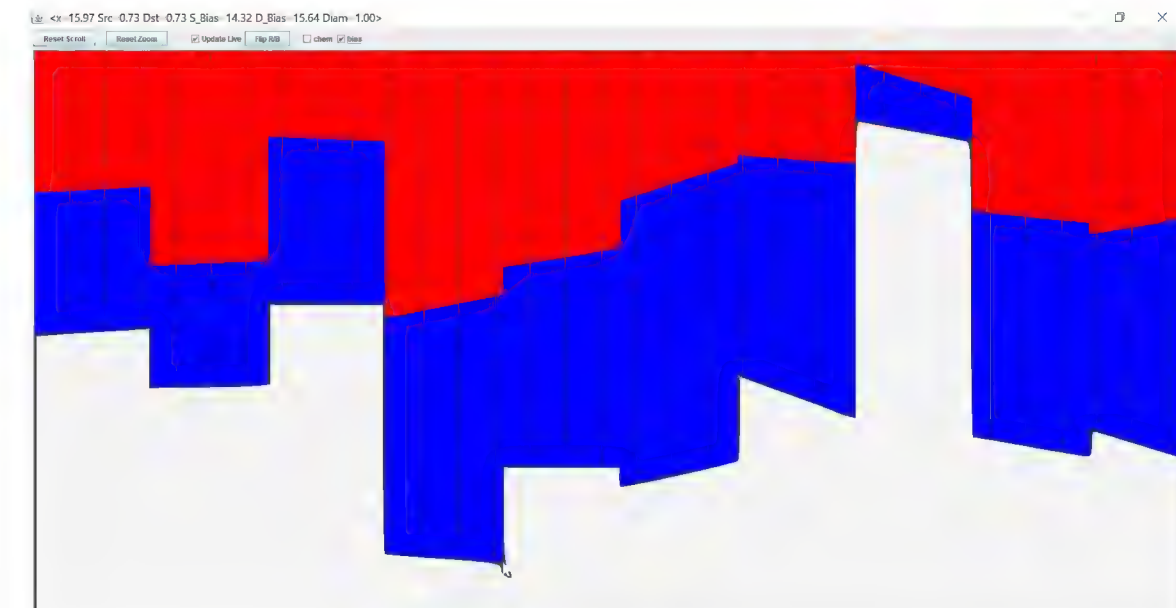
Figure 9.2: Bias Gradients for raw8, Individual 0, Generation 87. Values at the cursor are $Bias_{SRC} = 14.32$ and $Bias_{DST} = 15.64$.

9.6 shows the bias gradients for this individual.

If raw11, each individual was assessed on a range of initial altitudes, from 200 to 400 (by 10's). As can be see in figure 9.7, the performance is good throughout that entire range, except for a few curious spikes near 300. Figure 9.8 shows the bias gradients for this individual.
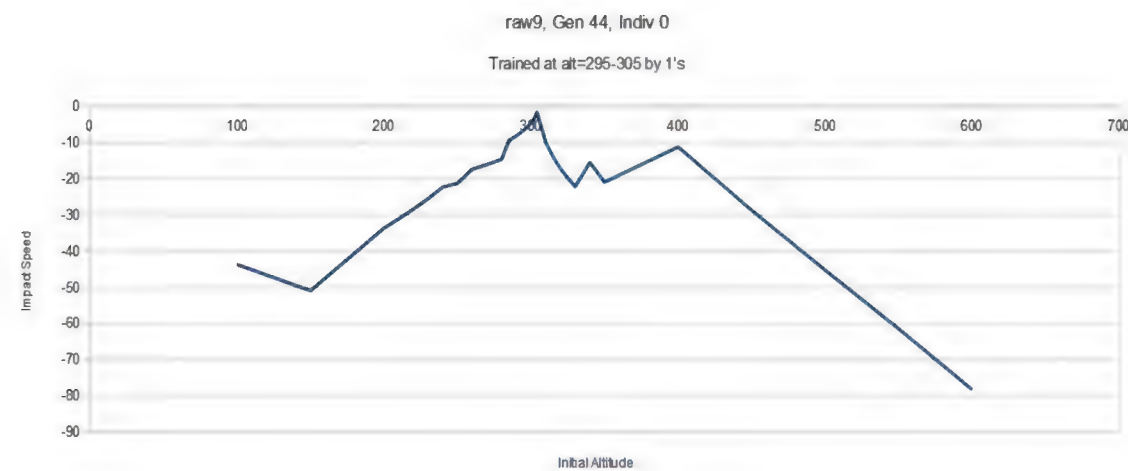
Figure 9.3: Impact Velocity vs Initial Altitude, raw9, Generation 44, Individual 0.
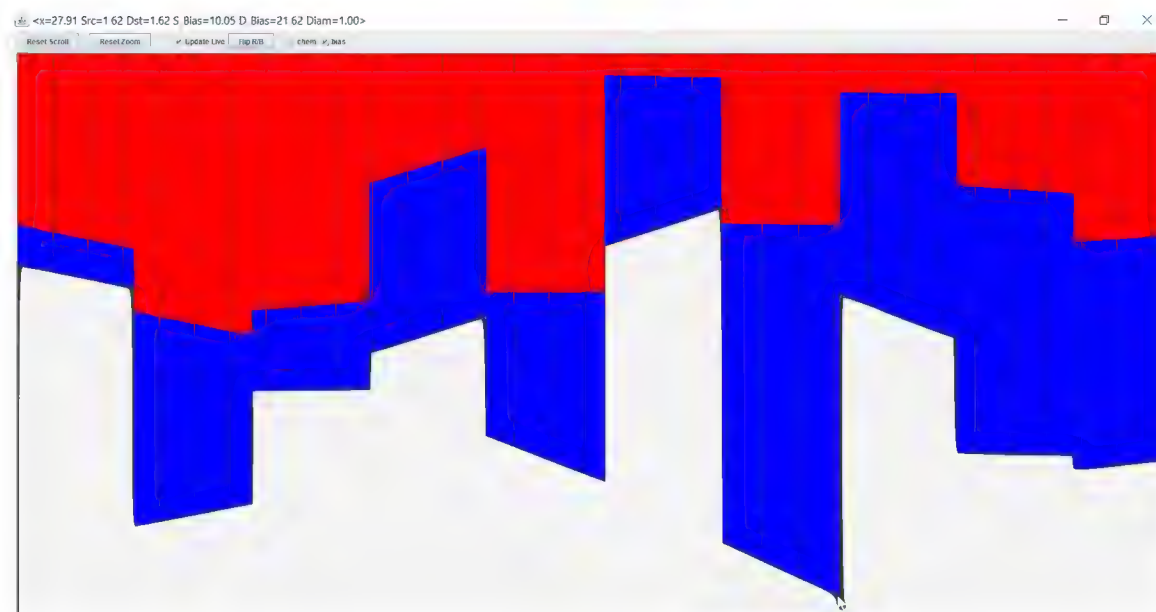
Figure 9.4: Bias Gradients for raw9, Individual 0, Generation 44. Values at the cursor are $Bias_{SRC} = 10.05$ and $Bias_{DST} = 21.62$.

/sectionDiameter Restriction

EA11HL (https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA11HL [11] contains similar experiments, except that the diameter of the input regions has been set to 0. With a diameter of 0, no chemicals can flow into or out of those regions. This does not however mean that those regions don't affect anything: the chemical amounts (offset by the biases) still dictate transfers from $SRC \rightarrow DST$; but the chemicals *which*
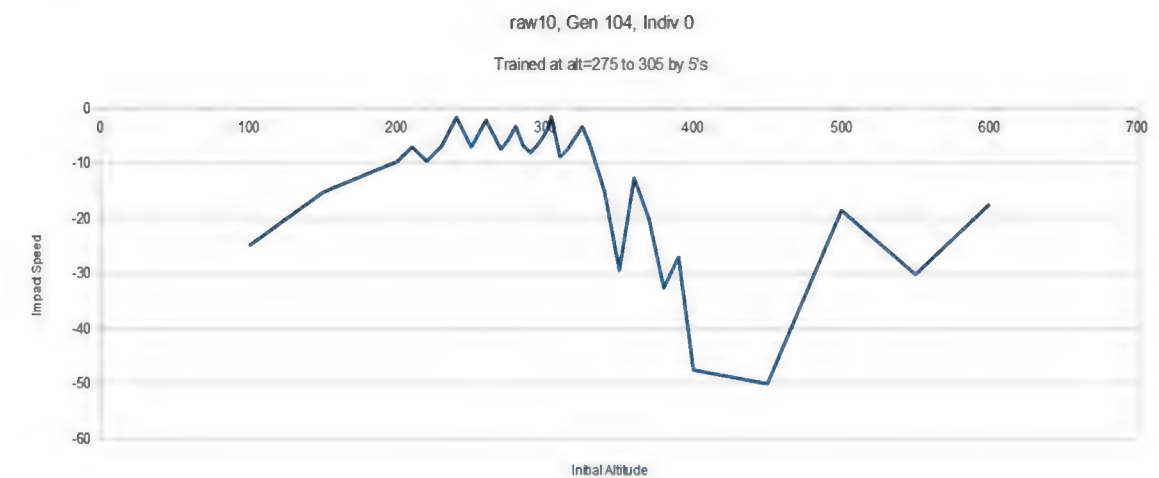
Figure 9.5: Impact Velocity vs Initial Altitude, raw10, Generation 104, Individual 0.
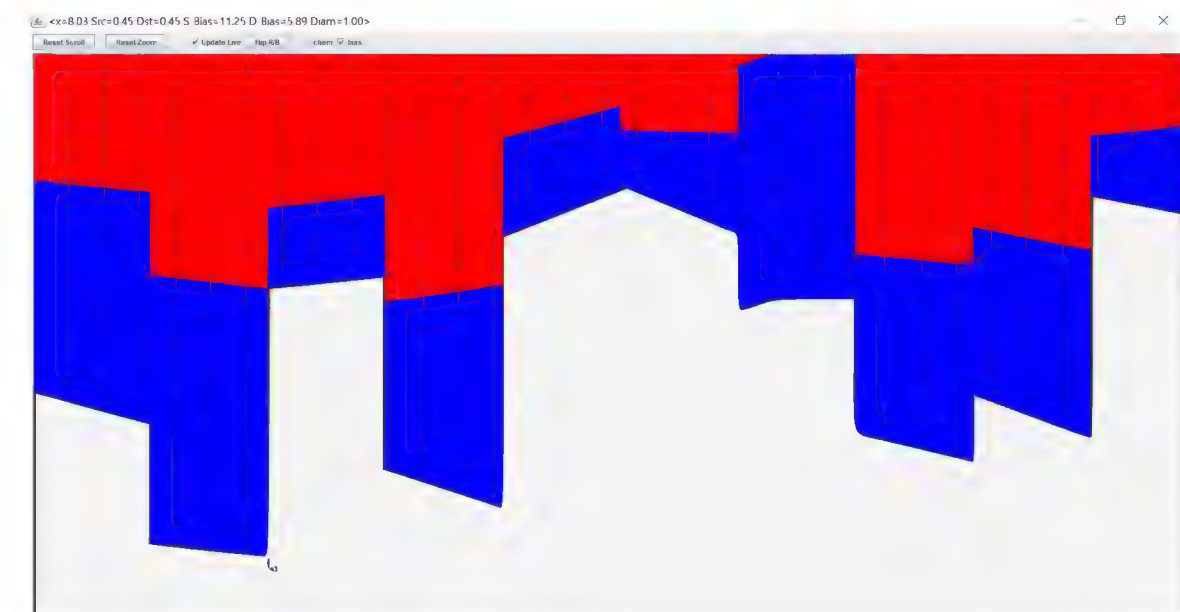
Figure 9.6: Bias Gradients for raw10, Individual 0, Generation 104. Values at the cursor are $Bias_{SRC} = 11.25$ and $Bias_{DST} = 5.89$.

code those instructions do not themselves change, and are never transferred anywhere.

This restriction prevents system saturation . Consider, for example, the fuel input region. At each timestep, the chemical levels in this region should reflect the amount of remaining fuel. Suppose though that somewhere there are transfer instructions that are removing chemicals from that region. The external system will continually add new chemicals to the system in order to set the fuel region's chemicals to the appropriate level.
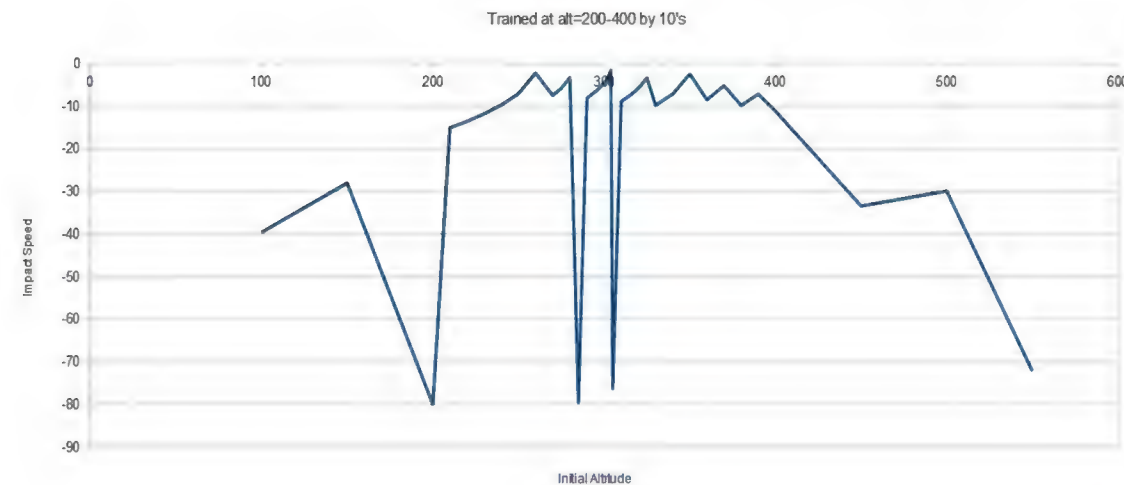
Figure 9.7: *Impact Velocity vs Initial Altitude, raw11, Generation 56, Individual 0.*
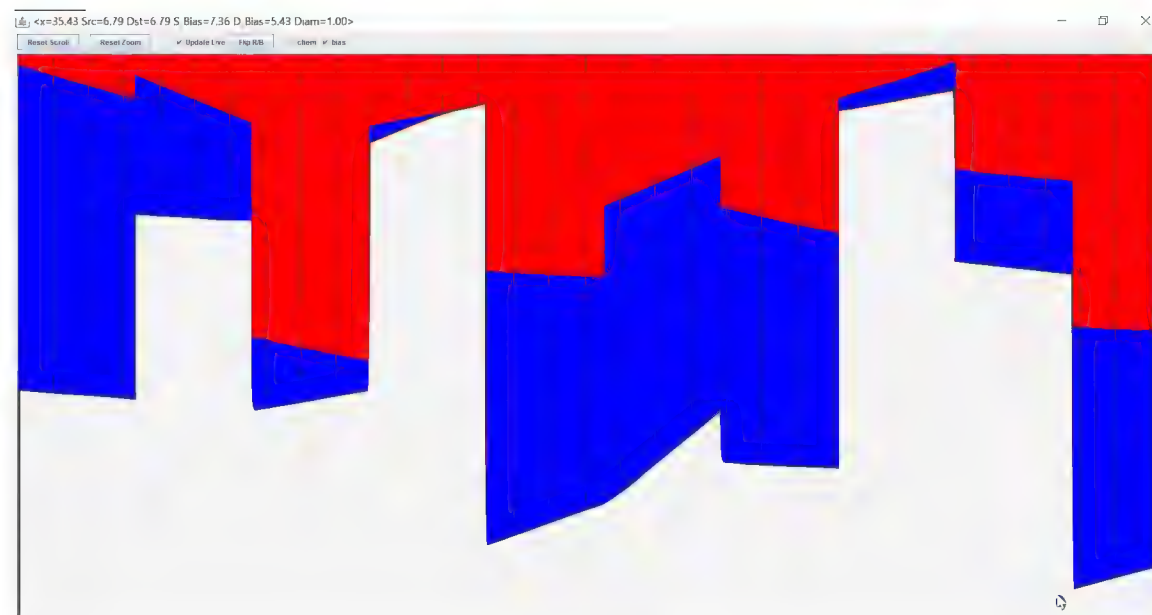
Figure 9.8: *Bias Gradients for raw11, Individual 0, Generation 56. Values at the cursor are $Bias_{SRC} = 7.36$ and $Bias_{DST} = 5.43$.*

As those chemicals are transferred to other regions, the total amount of chemicals in the system will increase. After a long enough run, the system may become so flooded with chemicals that it is no longer possible for it to function properly. Similarly, if chemical are transferred *into* an input region, then they will be removed from the system, and the memory may become chemical-starved. Restricting the diameter to 0 eliminates these

possibilities.

At first, it seemed that this diameter restriction made evolution difficult: but in fact, the system evolves as well as with the unrestricted diameters. See raw0diam, generation 141, individual 0, which was trained at a single initial altitude of 300. Figure 9.9 shows the behavior of individual 0, generation 141, when tested against different initial altitudes. As can be seen, the behavior degrades slowly below 300; but above 300, it drops off rapidly. In fact, the system's best performance is at an initial altitude of 304. At 305, it gets very close to the surface, and then engages the thrust fully, runs out of fuel, and free-falls back to the surface at a large impact velocity. This is the behavior at larger initial altitudes. In this case the "script" seems to be well-established: it's trying to drop around a distance of 300 with velocity=0 at the bottom. At an initial altitude of 305, it drops down to a minimum altitude just above the surface, but instead of landing, then engages the thrust until all fuel is spent, and then crashes to the surface.
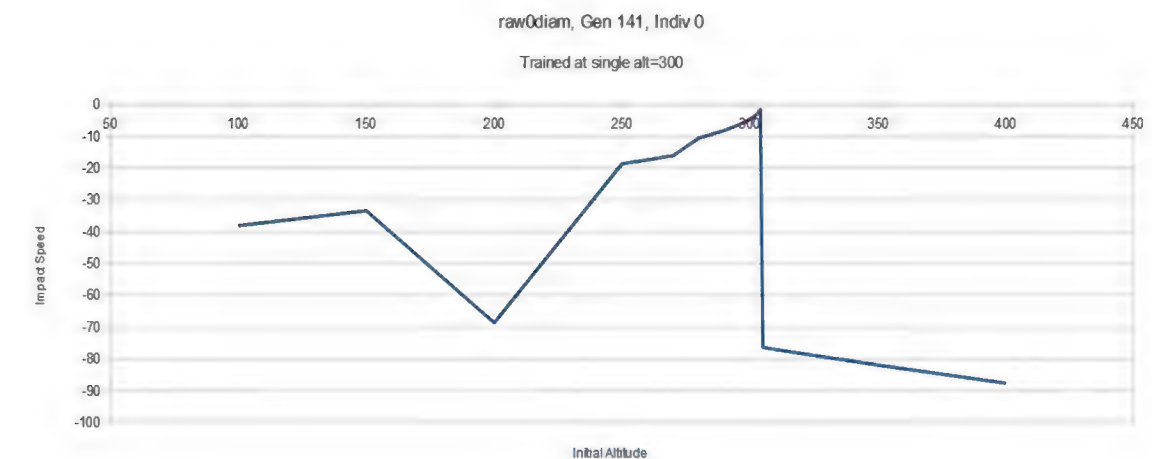
Figure 9.9: *Behavior of EEXIST-Controlled Lander with Input Diameters Set to 0.*

9.5 Conclusions

It does seem EEXIST is able to control a lunar lander game, though the experiments tended to wander into the "examination of GA dynamics" area rather than studying EEXIST *per se*. Nonetheless, it's interesting to observe that with the right set of bias gradients, the system can successfully land for a range of initial altitudes. As usual, there's no yet-apparent reason for the changes in the bias gradients across different training methods.

The system clearly looks non-linear, i.e., the performance doesn't always change smoothly with changes in initial altitude. It's also clear the system "doesn't understand what it is doing." It doesn't seem to engage the thrust in direct response to speed or altitude, for example. This is evidences by making small changes in the acceleration due to thrust a_T, which wildly degrades the performance of the system. Similarly, decreasing

the timestep causes the craft to come to a near stop at a higher altitude, and then free-fall with the thrust disengaged. This suggests the system may be essentially "following a script" rather than directly responding to input parameters (speed, altitude and remaining fuel). On the other hand, changing the initial amount of fuel (say increasing it, which should have no impact on the lander's performance) significantly changes the system's behavior: suggesting that the amount of remaining fuel *is* being factored into the system's calculations in some manner.

Part of the above can be explained by the limited training sets: 1, 11 or 21 different initial altitudes. The next set of experiments – the Virtual Eco System ("VEco") – will address this shortcoming, by continually training each individual.

9.6 Exercises

Exercise 9.1 Run the LL server and talk to it with telnet. ∎

Exercise 9.2 Run the analyzer and explore the different "raw" files (see README for details on how each file was evolved). Test these with the conditions they were evolved with, then test them with different initial conditions (fuel, altitude, initial speed). ∎

Exercise 9.3 Try evolving with very little initial fuel; can you get the system to evolve a more nuanced solution? ∎

Exercise 9.4 Try evolving with a variety of initial fuels, and see if you get a more robust system. ∎

10. EEXIST-Controlled Creatures in a Virtual Ecosystem
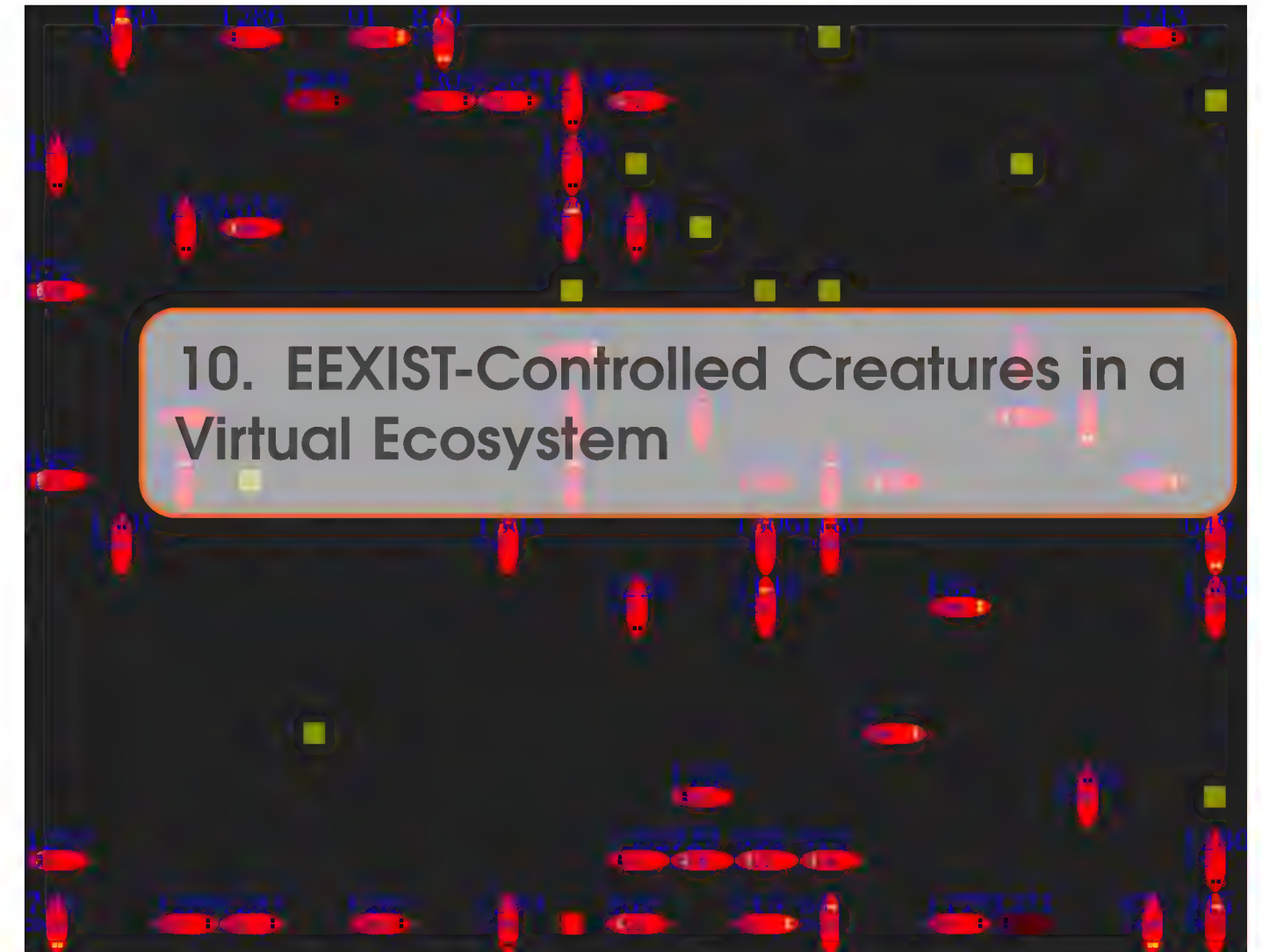
For an online synopsis of this project, see http://songlinesystems.com/VEco.html [25].

10.1 General Idea

The next set of experiments were based on the idea of a virtual ecosystem ("VEco"). The basic setup consisted of a virtual world in which creatures would:

- move around, expending energy as they move;
- absorb energy ("food") if they encounter it;
- attack other creatures; and
- mate with other creatures.

Each creature was controlled by its own EEXIST system. Information about a creature's immediate vicinity was supplied as inputs to EEXIST, and an output region was monitored to read EEXIST movement requests.

The setup was similar to the lunar lander game, in that a client/server model was

used. The server handled all bookkeeping on the environment (appearance/consumption of food), individual creatures (birth, location, energy level and death) and their interactions (attacking, mating).

Note that there is no need for a separate survival metric in this setup: survival *is* the metric. If a creature runs out of energy, it dies and is removed from the population. If it still has energy, it remains in the population, and may eventually mate, passing on its genes to a new generation.

Most action occurs when a creature moves forward. Movement onto an empty square simply changes the creature's location, and decreases the creature's energy by 1 unit. If this completely depletes a creatures energy, the creature dies.

Movement onto a square containing food increases the creature's energy. Movement into a square occupied by another creature has one of two possible effects:

- if the two creatures are facing each other, mating will potentially occur, with a new offspring appearing behind the creature that is moving;
- if the two creatures are *not* facing each other, the moving creature will attack the other creature, absorbing energy from it.

All code and outputs for these experiments are found on the VEco sub-directory, at https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/VEco [13].

10.2   Client/Server Setup

The server maintains an $n \times n$ grid of squares comprising the universe in which creatures exist. A simple command line interface allows a client to interact with the universe, using the following commands:

- R - reset the simulation
- B 1 or 0 - allow or disallow breeding
- E - deposit a random amount of energy on a randomly-selected square
- C - create a new creature. The server returns the new creature's integer ID
- id T R G B BL - set the color of the individual whose id is "id." RGB are the red, green and blue colors; BL is 1 for blinking, 0 for non-blinking
- id O - mark this individual as old (yellow eyes)
- id L or id R - indicate that individual "id" wishes to turn left or right
- id F - indicate that individual "id" wishes to move forward one square. The return value from the server is one of:
  - K id - the moved individual killed another creature, whose ID was id
  - M idparent idnew - the movement resulted in mating. idparent is the ID of the other creature involved in the mating; idnew is the ID of the new creature. Note that this information can be used by the client to merge parent genomes for the offspring.
  - OK - nothing special happened
- id Q - query the individual. The server returns a set of information about the individual and its surroundings (detailed below).

- D daylight - sets the daylight level of the system according to the value of "daylight" (0-25)
- Q - shutdown the server and exit

10.2.1   Query Response

The "Q" command asks the server to convey information about the area surrounding a creature. The response string is a single line, consisting of the following:

- the creature's current energy level (0 if the creature is dead)
- a single space
- 24 characters, describing each square in a 5×5 region centered at the creature. Possible characters are:
  - "-" for an empty square
  - "*" for a wall
  - "F" for a square containing food
  - "N," "S," "W" or "E" for a square containing a creature; the specific value indicates the direction in which that creature is facing.

The order in which this information is returned is shown in figure 10.1. The creature's position is represented by "C" on the middle of the region. In the return string from the Q command, the first of the 24 characters returned corresponds to the square labeled "1," followed by the character corresponding to the square labeled "2," and so on.

12	13	14	15	16
11	2	3	4	17
10	1	C	5	18
9	8	7	6	19
24	23	22	21	20

Figure 10.1: Ordering of Neighbors for Return String from the Query Command. The "Q" command returns the creature's energy, followed by a space, followed by 24 characters representing the state of each of the 24 regions shown. Information is presented for square "1" followed by "2" and so on.

10.3   Additional VEco Mechanics

The above description covers the basic movement, breeding and death of creatures within VEco, most of which are handled by the server. Beyond these details, there are many

variations possible, some of which were explored, some left for future work. Note that some of these are handled by the client code.

- Walls are currently impenetrable. A wrap-around model is another possibility, but this has not been explored.
- No energy is expended in turning, but a turn is always followed by a request to move forward. If the forward move is blocked though, the total energy expenditure remains 0.
- Mating is only allowed if both parents have at least a certain amount of energy.
- In some cases, mating is only allowed among parents who have exceeded a certain age.
- In some cases, mating can occur without parents coming into proximity of each other. At random intervals, new creatures are introduced into the population, but their genome is set to be a mix of the genomes of two randomly-selected parents.
- Creatures are processed in round-robin fashion, but newly-created creatures are always positioned at the head of the processing queue.

10.4   EEXIST Interface

Each creature has an EEXIST system associated with it. There is a client that keeps track of each individual creature (as does the server), and is responsible for communicating with the server. The client conveys information about a creature's neighborhood to its associated EEXIST system; steps EEXIST; and reads any requested action indicated by EEXIST.

The usual genome structure is employed, consisting of 10 equally-spaced regions, each with its own pair of SRC/DST bias gradients. The address space is split into five input and one output region, as follows:

- $[0, 4)$ input region R1 (figure 10.2);
- $[4, 8)$ input region R2;
- $[8, 12)$ input region R3;
- $[12, 16)$ input region R4;
- $[16, 20)$ input region R5; and
- $[24, 28)$ output region.

Regions R1-R5 are used to inject information about a creature's environment into EEXIST. R1, R3 and R5 describe the state of the corresponding regions shown in figure 10.2. For each of these regions, the following SRC and DST chemical levels are set:

- $SRC = DST = 5$ is the square is empty;
- $SRC = DST = 25$ is the square contains a creature that is facing you (and thus is a potential predator or mate, depending on which way you're facing); and
- $SRC = DST = 15$ otherwise (square contains a wall, food, or a creature that is neither a threat nor a potential mating partner).

Initially, R2 and R4 conveyed further neighborhood information; but these were later changed to allow input of more-general information. Region R2 is a general-area input. For the 24 squares around the creature, the number of squares that are not empty (i.e.
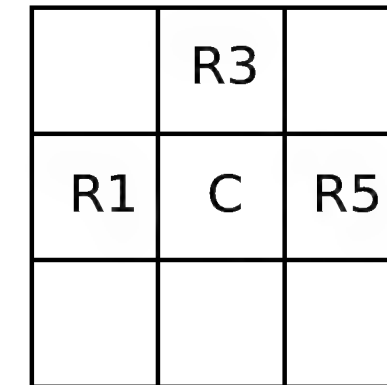
Figure 10.2: Input Regions for VEco. For a creature centered at "C," information about R1, R3 and R5 are sent into the creature's corresponding EEXIST system.

contain a wall, food, or a creature (in any orientation)) are counted, and SRC and DST in region R2 are set to that count. Thus, R2 shows, roughly, how crowded the area is (though that crowding could be beneficial, dangerous or benign).

In the initial runs of the system, creatures tended to move to the edges of the universe and sit there. If all inputs are based on the contents of the surrounding squares, and nothing is moving, then none of those inputs ever changed, and thus creatures that stopped moving (often, but not always, because they were facing a wall) would never start moving again. This might be rectified by having energy levels drop over time, while also giving a creature information about its own energy level. This however felt a bit "rigged." Instead, a new input was introduced: a cyclic "daylight" variable. This variable runs repeatedly from 0 to 25 and back to 0, changing every 50 ticks of the simulated universe (one tick per creature update). In region R4, $SRC = DST = daylight$. This helps keep the system from becoming stagnant, by changing the inputs to a creature even if the creature is not moving and nothing is changing in nearby squares.

There is a single output region defined at $[24, 28)$, which is interpreted as a movement request by the creature. The average $SRC + DST$ is calculated for this region, and used to determine an output as follows:

- $5 \leq SRC + DST < 10$ turn left and move forward;
- $10 \leq SRC + DST < 15$ turn right and move forward;
- $15 \leq SRC + DST$ just move forward.

10.5   Experiments

As usual, a number of experiments were run, with a lot of variations in the experimental setup. Throughout the run, genetic information was written to a file, as was a synopsis of birth and death events. The general setup for an experiment was to initially seed the population with a set of individuals with randomly-generated genes, and then allow them to interact throughout time. The server presents a graphical display of the universe, such

as shown in figure 10.3.
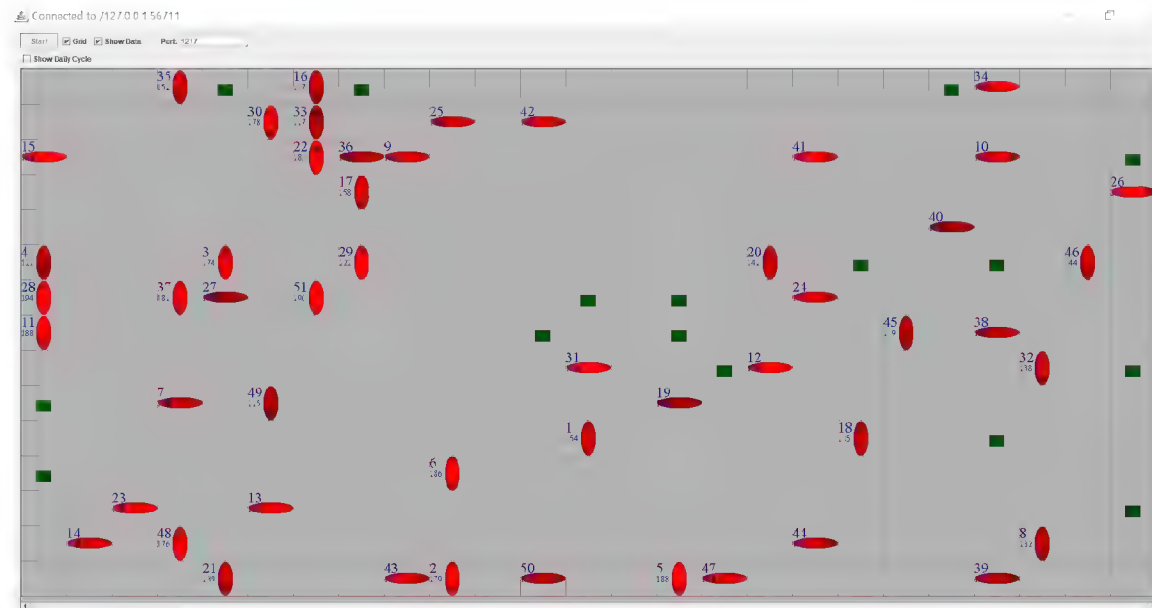
Figure 10.3: Sample Graphical Server Output. Creatures are shown in red; eyes are drawn to indicate which way the creature is facing. Green squares represent food. ID numbers are shown near each creature, and the creature's energy level is shown below the ID (smaller text). The intensity of the creature also correlates to its remaining energy: creatures with more energy are drawn brighter than those with less energy.

The client has a user interface, where command-line instructions can be given (note that this is different from the command-line interface to the server, which the client control). The following commands are available:

- ? - display a help message
- t id - tag an individual (default coloring)
- t id R G B - tag an individual with the given color
- e id - show the general EEXIST display window for the given individual
- e2 id - show the detailed SRC/DST window for the given individual
- f filename - open the given filename for reading genetic information
- c src dst - clone genetic material from individual "src" to individual "dst"
- breed on - enable breeding
- breed off - disable breeding
- pause - pause the simulation
- run - resume the simulation
- d - inject a drone (see the subsection below)
- @filename - run commands from the named file
- reset - reset the simulation (client and server)
- #anything - create a comment (not interpreted)

- Q - quit immediately

In addition to a text/command input area, the client's control panel includes sliders for adjusting:

- the system karma (this affects the karma of all existing individuals, as well as creatures created in the future);
- the initial population size;
- a delay between updates (to make it easier to see what's happening int he graphical output); and
- the number of EEXIST timesteps used to update one creature before moving on to the next creature.

There is also a checkbox for pausing the system (not that the state of this check box is not affected by the pause and run commands), as well as buttons for resetting the system and for exiting the simulation.

10.5.1 Drone Interaction

After letting a population of individuals develop and evolve, it seemed like it might be useful to be able to interact with the population, so a *drone* capability was added to the system. The "d" command requests injection of a drone into the population. The drone is highlighted, and can be maneuvered with the f (forward), l (turn left) and r (turn right) commands. Note that unlike other commands, these keystrokes do not need to be followed by ENTER: they are single-key commands (but holding the key does not successfully register as multiple key presses). This makes it relatively easy to manipulate the drone inside the simulated ecosystem, allowing the user's creature to eat food, to approach and attack other creatures, and so on. The "q" key is used to exit the drone-control mode, and return to the regular command line interface (where ENTER is required to execute a command). Upon leaving drone-control mode, the drone remains in the population, but does not move nor mate.

It is difficult to meaningfully quantify the behavior of the population in response to drone actions, but subjectively, it appears that the population *does* respond to the drone's movements. In some cases, the older creatures seemed to move away as he drone approached. This behavior was interesting enough to encourage further experimentation, as described in the next sections.

10.5.2 Longevity Data

As the system runs, new creatures are given sequential ID numbers, thus the ID shows relative age of individuals (creatures with higher IDs being younger than those with lower IDs). An initial analysis of the population can be made by looking at the distribution of individuals' ages.

Figures 10.4-10.11 show graphs of age vs. ID number at different points during the development of the ecosystem. Figure 10.4 shows the population at 100,001 cycles. The population has some individuals with an age around to 100,000 (these are likely first-generation individuals), as well as a number of younger individuals (those with IDs above

100).
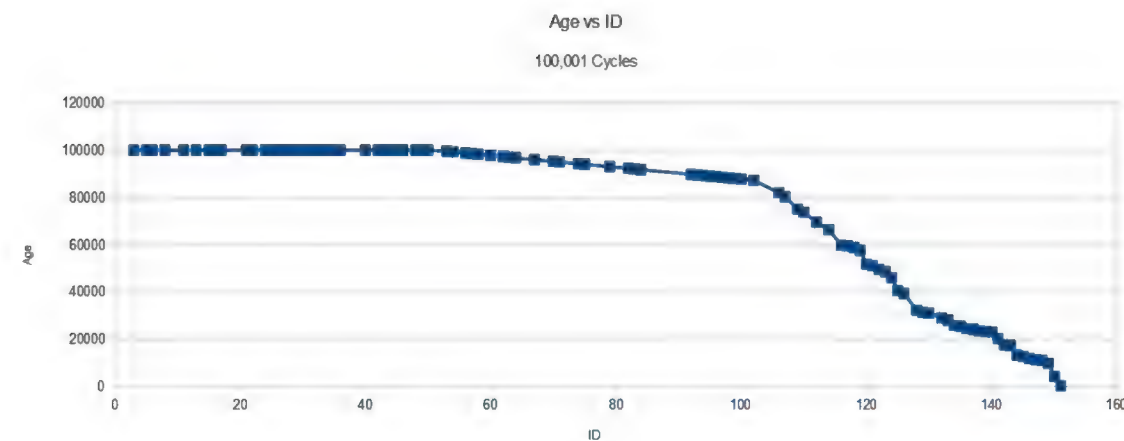
Figure 10.4: Graph of Individual IDs vs. Age, Cycle 100001. A number of members have survived from the beginning of the run.

Figure 10.5 shows the system at cycle 200001. The older individuals remain, but there are more younger creatures. This trend continues through figure 10.8, at cycle 1000001.
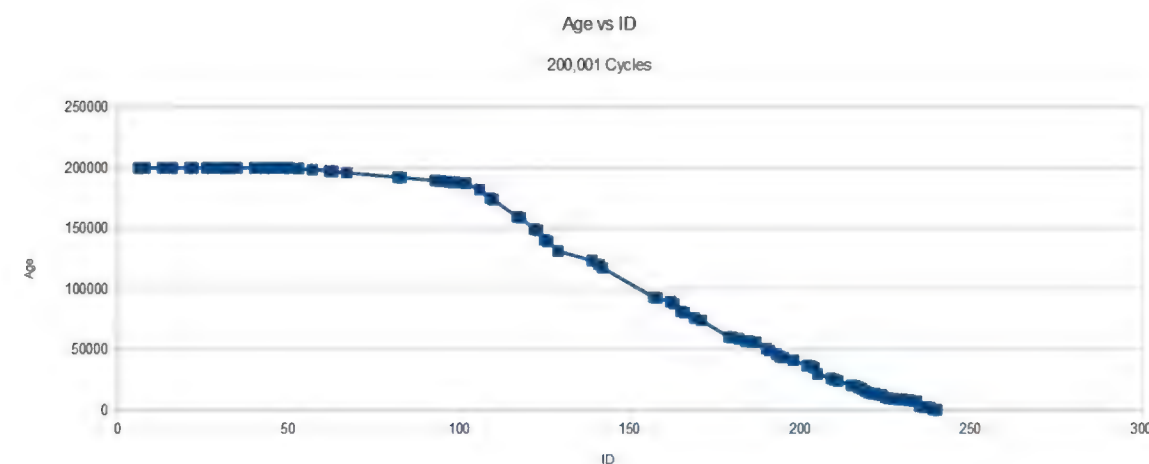
Figure 10.5: Graph of Individual IDs vs. Age, Cycle 200001.

If figure 10.9 (cycle 2000001), the oldest individual is younger than 2000001 cycles. It appears the previous longest survivors have died, and the age distribution of the remaining population is becoming linear.

In figure 10.10 (cycle 3,000,001), the oldest individual has an age below 1,000,000, and the age distribution looks linear. Figure 10.11 shows the system at cycle 6,000,0001. The oldest individual has an age below 500,000, with a mostly linear age distribution.
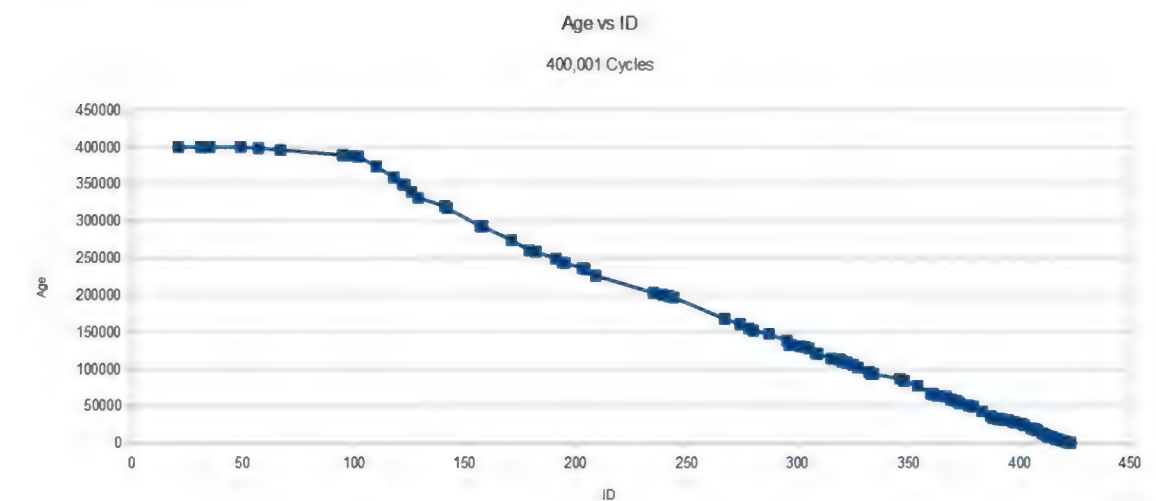
Figure 10.6: Graph of Individual IDs vs. Age, Cycle 400001. A lot of the original population has died.
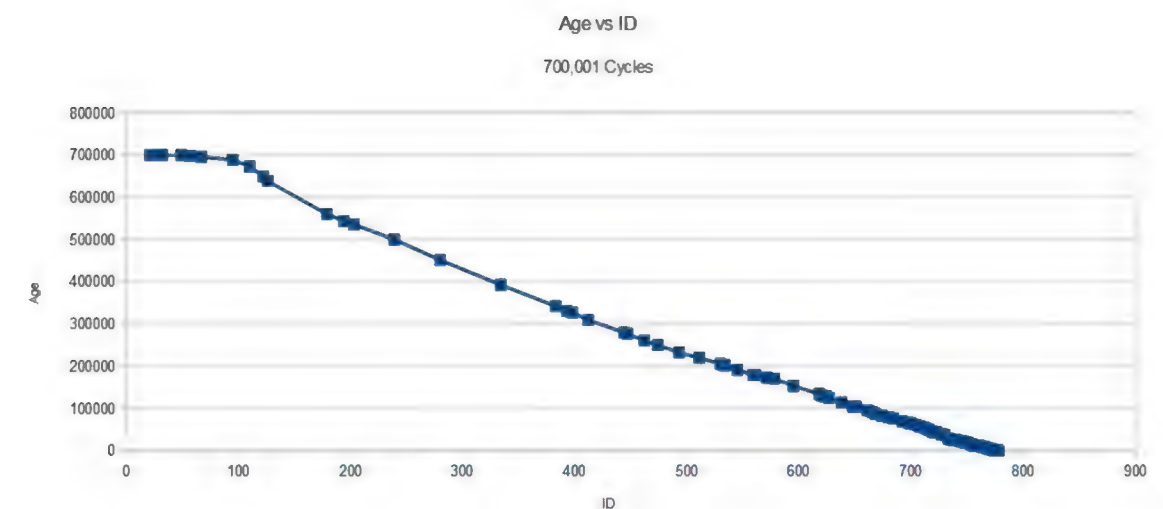
Figure 10.7: Graph of Individual IDs vs. Age, Cycle 700001. Most of the population's ages fall in a linear distribution.

The trend seems to be that as the system ages, the oldest individuals get younger and younger (this was observed in multiple tests). There are (at least) two possible interpretations of this:

1. random individuals in the original population happened to be well-suited to survive, but only against the rest of the initial population. As more and more individuals came into the population, the advantages possessed by these first-generation individuals
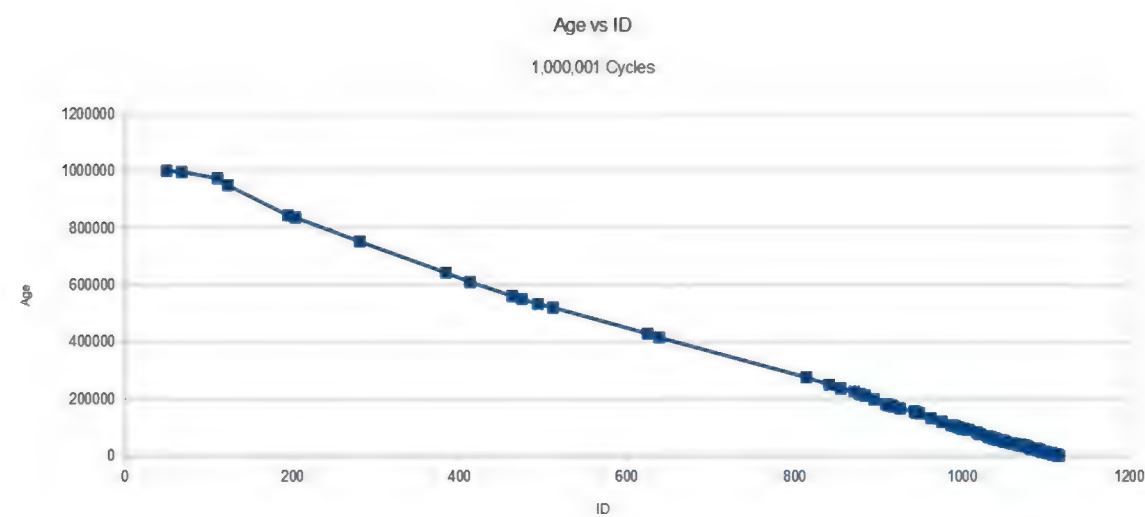
Figure 10.8: Graph of Individual IDs vs. Age, Cycle 1000001. Most of the population is younger than 400000.
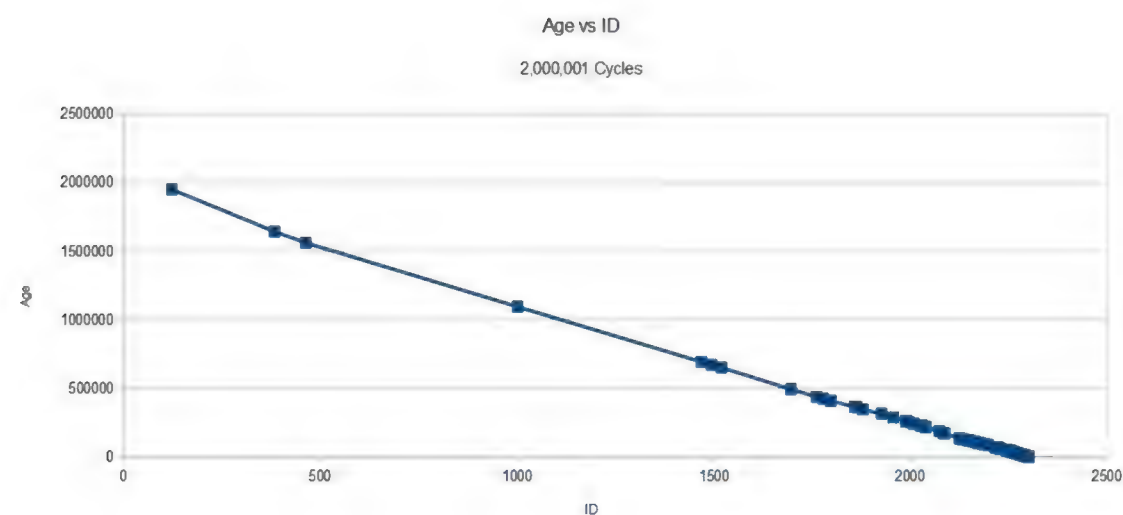
Figure 10.9: Graph of Individual IDs vs. Age, Cycle 2000001.

turned out to be nothing special; or

2. the population really *is* learning to survive better, and over time, the entire population has developed survival mechanisms, thus giving none of the individuals any particular advantage over the others.

The most interesting explanation would be #2. In order to test this, a new experiment was designed: namely, testing a trained population against a randomly-generated population.
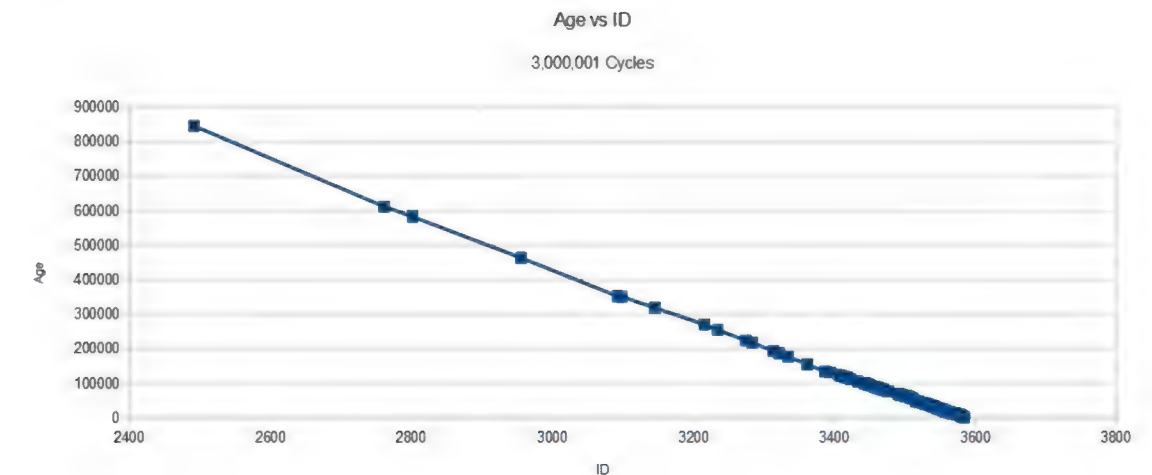
Figure 10.10: Graph of Individual IDs vs. Age, Cycle 3000001. All first-gen members have now died; the oldest members of the population are aged around 850000.
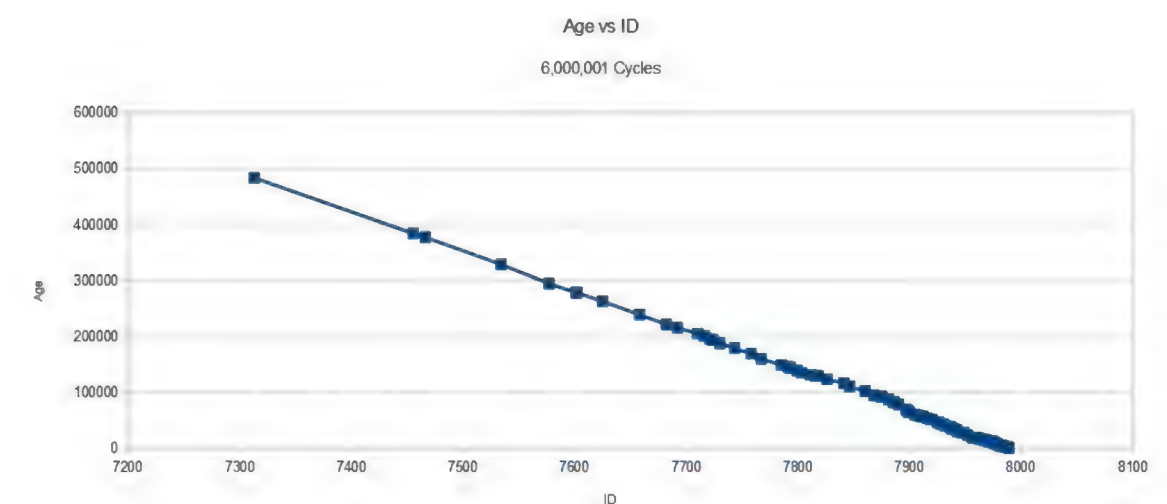
Figure 10.11: Graph of Individual IDs vs. Age, Cycle 6000001. The oldest members are aged under 500000, while most of the population is under 200000.

10.5.3 Trained Vs Untrained Population

The goal in this set of experiments was to test the above hypothesis: namely, that a trained population would survive better than an untrained population. To test this, a population of 50 random creatures was created. Then the 25 oldest creatures from an aged population (the one that was 6 million cycles old, from figure 10.11) were cloned into 25 members of the random population. Mating was turned off, and the ecosystem was simulated, allowing the creatures to interact. The results were consistent across multiple experiments: almost

all of the initial deaths were from the young/random population of creatures, while the old population decreased far more slowly. Figure 10.12 shows the decline in the population across time. While the general trend of these graphs is in line with what was anticipated, some of the artifacts are still confusing: for example, why deaths seem to occur in clusters of 3 or 4 at a time.
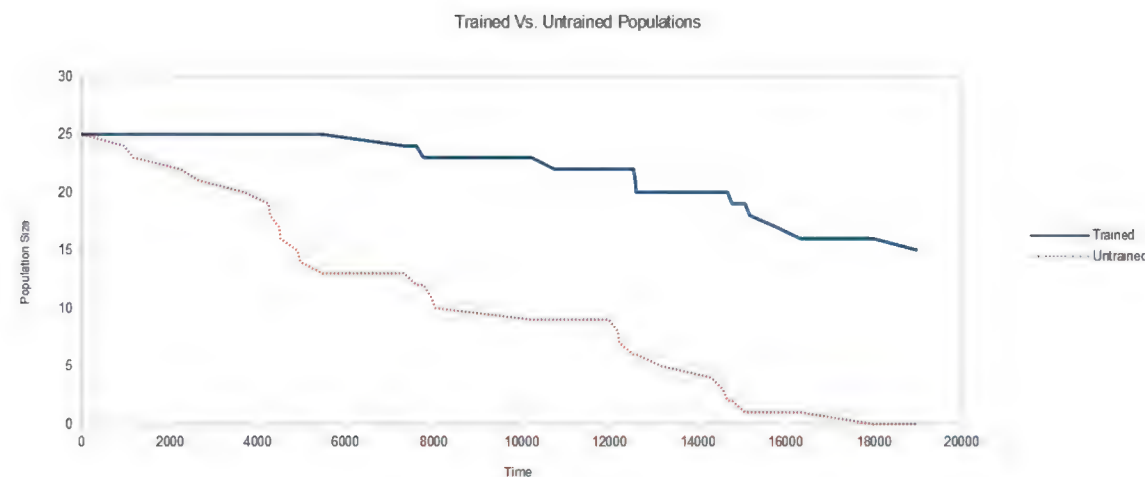
Figure 10.12: Injection of 25 Trained Creatures Into a Population of Random Creatures. As can be seen the trained creatures survive better than the untrained ones. By the time the first trained creature has died (after timestep 7000), nearly half (12 out of 25) of the untrained population has died. At timestep 14680, 80% of the trained population is still alive, while the untrained population is down to 8% (2 survivors out of 25).

10.6   Exercises

> **Exercise 10.1** Run the server (VEco.jar) and control it with a telnet connection.

> **Exercise 10.2** Run the server and main program's .jar files (available here [13]) and evolve a population.

> **Exercise 10.3** Inject a drone into the ecosystem and explore how the population behaves.

> **Exercise 10.4** Vary karma and repeat the above.

> **Exercise 10.5** Change the number of cycles per step and re-evolve.

> **Exercise 10.6** Try working with different sized universes.

> **Exercise 10.7** Change the code to consume energy whether a creature is moving or not.

Part Three - Recent Work

Timestep: 4661 (Running)
Displaying Src chemical levels
Loc:(0.23,13.23) SRC: (30.84,35.23) DST: (24.79,10.27)

11. 2-D Spatial Domain

The above work is all built around a 1-D notion of space: all tubes of chemicals are arranged along a single axis, and addresses refer to a single displacement along that axis ("x"). There is reason to believe (or at least suspect) that a 2-D system could have inherently more capabilities. This is in part because of the richer space of connections/effects between nearby points vs. what is available in 1-D simply by adjusting bias values.

11.1 2-D Setup

Experimental work has been done incorporating *four* different chemicals:
- Source X - the x coordinate of the location from which chemicals are to be drawn
- Source Y - the y coordinate of the location from which chemicals are to be drawn
- Destination X - the x coordinate of the location to which chemicals are to be transferred

- Destination Y - the y coordinate of the location to which chemicals are to be transferred

The underlying behavior of the system is the same as in the 1-D case: chemical levels at each location specify a transfer from a source to a destination. The only change is that the source and destination addresses now each require a pair of coordinates.

11.2   2-D Karma

Recall that in the 1-D case, karma (κ) represents a spreading of the effect of each transfer request. With a karma of, say, 5, a single transfer from x to y will also cause a (smaller) transfer from $x \pm \Delta$ to $y \pm \Delta$ where $0 < \Delta \leq 5$.

In the 2-D case, there is a *2-D (circular) region* around the source and destination point of transfer. κ is the radius of this region, and as in the 1-D case, the effect of a transfer tapers off linearly with respect to distance form the center of the circular region, scaled by κ.

This feels like a natural extension of karma from 1-D to 2-D; unfortunately, it greatly increases the computational load on the simulator: doubling the resolution in x and y causes a 4x increase in the number of transfers to calculate; but due to the karma, for each transfer there is now an additional 4x increase in the number of nearby karmic transfers. This is thus an $O(n^4)$ process.

This was a hindrance to experimentation, as the 1-D simulation was already leading to simulation times of weeks for most experiments. Fortunately, Peter Athanas (Professor, Bradley Department of Electrical and Computer Engineering, Virginia Tech) offered me use of a 28-core Intel ® Xeon ® Platinum 8176-based machine. This allowed execution of 25 threads in parallel, for an almost perfect 25x speedup in simulation time.

11.3   Image Recognition

A natural application for a 2-D EEXIST is image recognition (this was in fact one of the motivating problems for this extension to 2-D). Work began Sept 2018, with the following basic scheme:

- use an image's pixel values to set chemical levels in a 2-D region of tubes;
- let the system run for a number of time steps; and then
- compare chemical levels in two or more regions, and use the comparison to determine the system's "choice" in response to the initial image.

This is, incidentally, a deep application of code/data non-dualism: the pixel data is directly controlling the processing of the pixel data.

Using this approach, the system was trained on a set of images, after which it was presented out-of-sample data which it tried to identify. As usual, there was nothing inherently image-processing-related to the setup. Instead, the system was allowed to process the data (using the data as the source of processing commands) and evolve to, hopefully, exhibit a desired capability.

11.3.1   Experimental Setup

The specifics of the experimental setup are fairly random. With no particular insights about how best to setup this system, the details were hacked out on a cross-country plane ride, simply to provide *some* starting point. Very little work has gone into modulating this initial setup; instead, with this arbitrarily chosen setup, most work has gone into exploring its capabilities.

Work was done using the Yale Face Database (P. Belhumeur, J. Hespanha, D. Kriegman, *Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, July 1997, pp. 711-720). This database contains images of 15 different subjects, each in one of 11 different poses (e.g. normal, with glasses, surprised, sleepy, etc.).

Each image in the database is 320×243 with 24-bit pixel values (8 bits for each of red, green and blue). For each pixel, the 8-bit values for red, green and blue are averaged, giving a color-agnostic *intensity* value (α) for each pixel. These are scaled by $1/7$, giving a maximum intensity of around 37 (chosen since the X and Y dimensions of the EEXIST address space is 40×40).

An image is scanned at $1/4$ resolution (every other pixel in X and Y), and the intensity value for the pixel at (x,y) is used to set chemical levels at location $(x/16, y/16)$ in the EEXIST address space. The chemical levels are set as follows:

Chemical	Initial Level
Source X	α
Source Y	$\alpha + 5$
Destination X	$\alpha - 7$
Destination Y	$\alpha/7$

This loads a copy of the image into the EEXIST address space in the upper-left region (see Figure 11.1).

All other tubes in the system are initially empty.

Once the system is seeded, the simulation is stepped a fixed number of steps (typically 250), after which the total amount of chemicals in each of a series of output regions (S1-S5 in Figure 11.1) is determined. The region containing the most chemicals is taken to indicate the system's identification of the initial image.

The system was trained using the usual genetic algorithm approach described previously (Part Two of this text). Each individual was presented an image from the database, the simulation was stepped, and points were awarded based on the accuracy of the system's determination. This was done across a pre-determined set of training data. Each individual thus obtained a score, which was used to mate/propagate individuals from one generation to the next. At some point (usually based on the scores of the most-fit individuals), out-of-sample images were presented to the individuals, and their scores were noted.

11.3.2   Experimental Results

A number of experiments were performed using this setup. Each experiment involved training the system to identify two or more subjects, regardless of the subject's pose
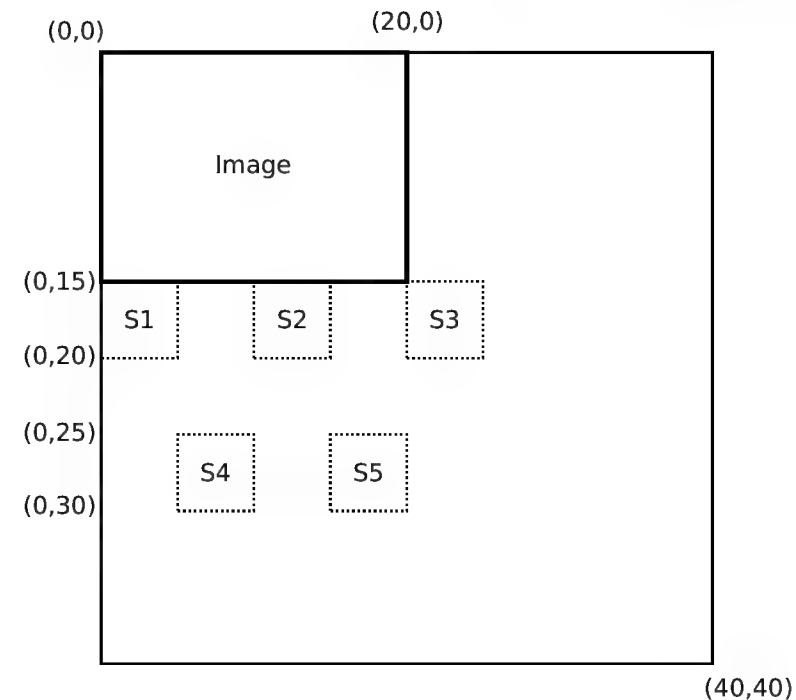
Figure 11.1: EEXIST Address Space Regions for Image Recognition Training and Analysis.

(smiling, sad, etc.) Training usually took from a week to a month, depending on the number of generations, value of κ, and so on. Most runs were spread across parallel 25 threads , each consuming close to 100

In most cases, the system was able to evolve individuals that did a perfect or near-perfect job of identifying subjects from the training data. The system's performance on out-of-sample data was generally lower, though sometimes still very good. As usual, there was very little rigor in the choice of experiments: it was mostly ad-hoc, poking around the design space and seeing what happens.

It turns out one of the earlier experiments (Run 3) gave surprisingly good results, which are detailed below.

11.3.2.1 Run 3

Experiment 3 (started 14 Sept 2018, finished 20 Oct 2018) used a $\Delta x, \Delta y$ of 0.125; max x,y of 40; and a karma (κ) of 1.0. The system was trained on two images (subject 01 and subject 02), with 5 poses each (glasses, happy, leftlight, noglasses and normal). See Figure 11.2.

After 23 generations, one individual (7) reached a score of 199/200. This individual was then tested on out-of-sample data (rightlight, sad, sleep, surprised and wink) and scored perfectly on all 10 images (Figure 11.3).

11.4 Evolving a Video Game Controller

In 2018, I gave a talk at Clark College, where I discussed the ideas behind EEXIST, the current state of the project, and threw out an invite to students to let me know if they were interested in working on a project in this area. Students in their final year usually undertake a Service Learning Project (SLP) of their own choosing. Occasionally a student will express interest, but it's rare that they actually put in the time and effort required to turn that initial interest into a SLP.

Following my talk though, I had one student approach me about working on this project. He had a powerhouse gaming system with a massive GPU (2500 cores), and was interested in porting EEXIST to run on a GPU. It sounded very ambitious, but I gave him the guts of the Java simulator, outlined the main algorithms, and suggested he explore from there. To my surprise (and delight), he announced a few days later that he had ported the system, and it was running well and showing considerable speedup over the Java-based simulator.

Despite a number of technical issues, he managed to grow this to use thousands of cores in parallel, and then set about evolving a system for controlling a car in a video game. The speed of his system was staggering; and porting my own work to a hefty GPU remains on my long-term To-Do list.

Appendix A contains a copy of his final SLP report, which also includes a number of links to different parts of his work.

Training Data: 5 poses each subject
(Yale Face Database)
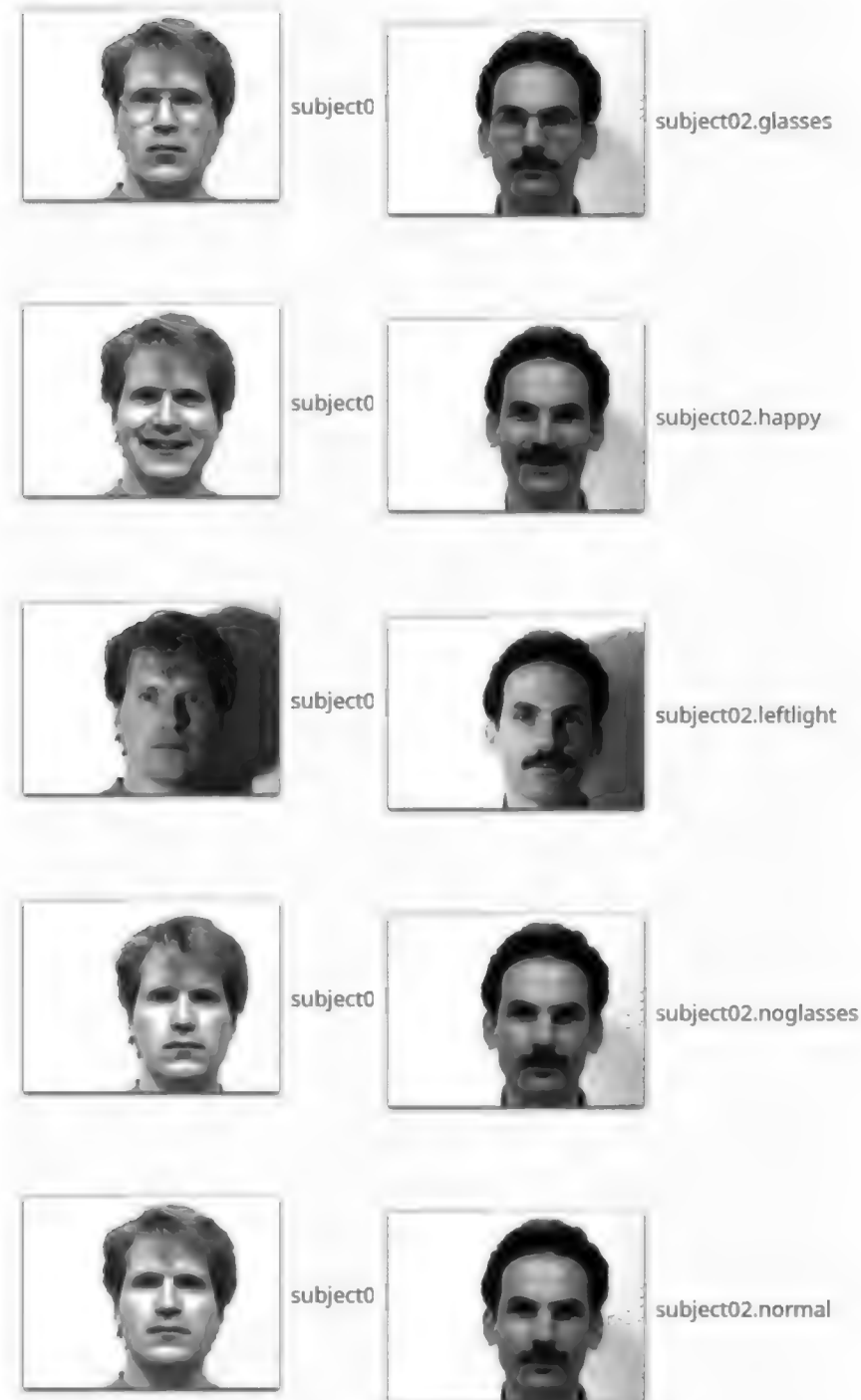
Figure 11.2: Training Data for Raw 3

Training Data: 5 poses each subject
(Yale Face Database)
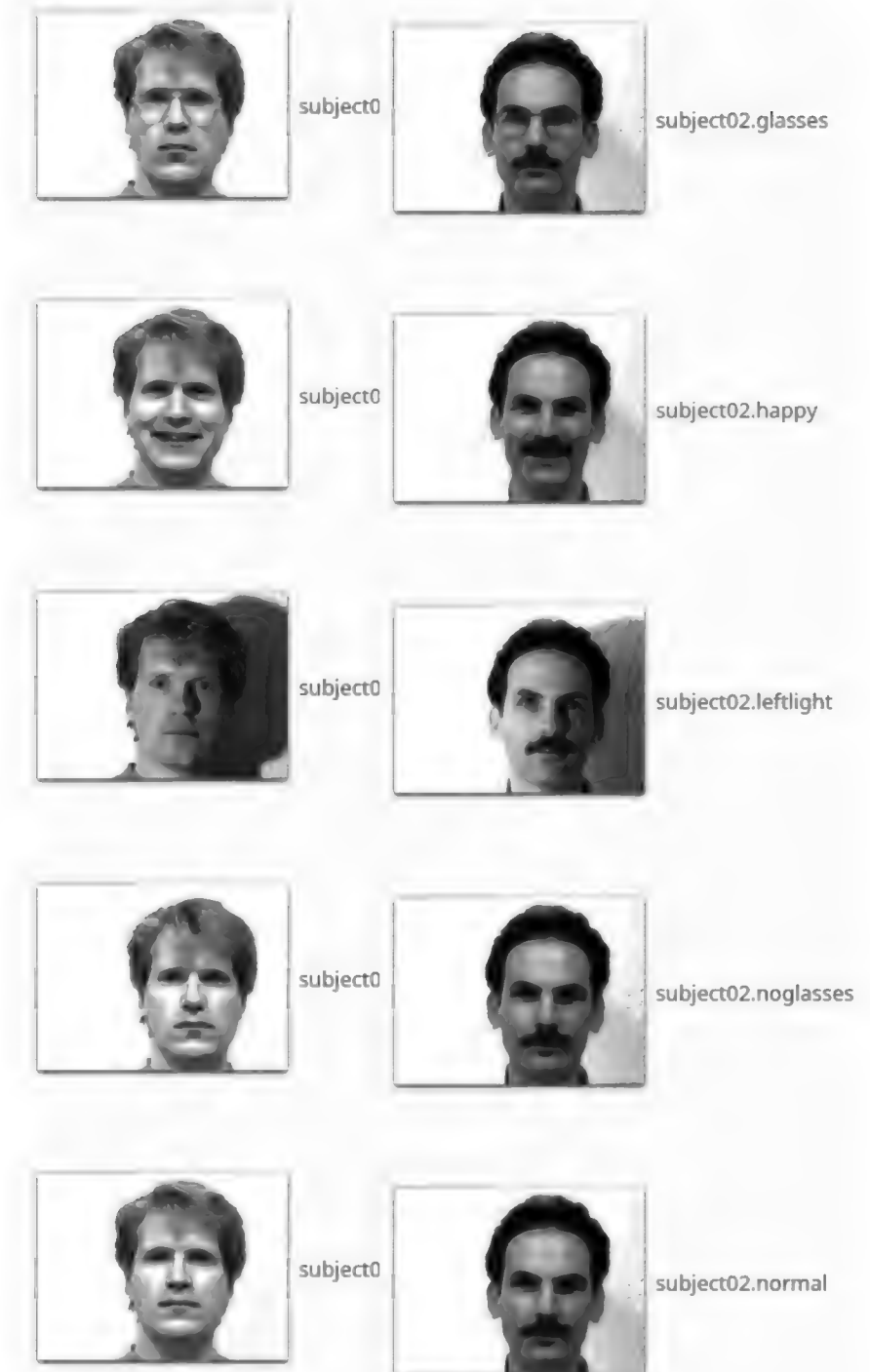
Figure 11.3: Out of Sample Data for Raw 3

12. Free Will Injection

In Spring 2020, a question arose regarding the behavior of EEXIST-controlled individuals in a virtual ecosystem (Chapter 10). The question is one of free will, or of freely making a choice. While this and related subjects (determinism, randomness, causality) have been studied for thousands of years, it is not my goal here to explore the question of free will itself *per se*. Rather, I want to use the notion of free will to drive an exploration of different parts of the design space for EEXIST. So let us ponder this question, understanding that nothing in this pondering is original, nor definitive, but that these thoughts are simple guideposts to a different aspect of EEXIST.

Note that this work was done on the earlier 1-D EEXIST system, not the 2-D system described above.

12.1 Essence of Free Will

Okay, so here's a question: what (for humans) is the essence of free will? If one imagines that we are just machines, and our brains are following programs, and based on all of the inputs, history, and our current situation, we are going to make a best decision, it feels like there is likely a single best decision to be made. If having (and exercising) free will means

one is free to make a different decision, does this suggest that one might make a decision other than that best decision? if so, that would seem to border on almost some kind of *malfunction*.

Put differently: imagine you're given all the information about the current situation you're in, every aspect of the universe, all the history and details that you might possibly require to decide what to do in this moment that is in your best interest. And now imagine that you decide to do something different. What just happened? There are a few possibilities:

- It could be that the decision you did make was in fact the best possible decision, because, for example, you were trying to prove that an oracle was wrong when it said you were about to lift your left arm and so you decided to lift your right arm instead to prove that this oracle was a phony. In that case, even if all the other inputs would have said you should lift your left arm it ended up being in your overall best interest to lift your right arm instead. And so, even though it feels like that was a freely made choice, it was in fact still the outcome of a logical process.
- Another possibility is that you're simply malfunctioning: you're acting in a way that is contrary to the outcome of what your program says you should do. This would somehow relate free will with a malfunction, which feels, well, a bit weird to me.
- Yet another possibility is that there is no knowable "best possible choice" *given the information you have*. Picking lottery numbers, for example, is a situation where, given all available/usable data, there is little guidance on which numbers would be best to pick. But even in this case, is there a well defined algorithm at work, guiding your choice of numbers? The fact that we don't know/don't understand that algorithm is irrelevant here: the question is, is your "choice" in fact dictated by some algorithm, in a way that given the same initial conditions, you would make the same choice?

12.2   Free Will and Randomness

It's from that third possibility above that I see an opportunity for non-deterministic choices to be made. If you need to choose between two options, and there is not enough data to make one option even slightly preferable over the other, then a *random* choice may be appropriate. Assuming this is a truly random (vs. pseudo-random) choice, the result could be seen as non-deterministic.

12.3   How Can Free Will Make Sense?

Suppose you always walk to work on certain streets, chosen because of their low traffic, short distance, pleasant scenery: your clear best choice of what path to take from home to work. And suppose one day, out of the blue, a bus loses control, drives onto the sidewalk, and kills you. Nothing in your available data would have warned you about this. You had no foreknowledge of the mechanical state of the bus, and the incident happened too quickly for you to respond as data became available. Your choice to follow your usual

path to work that day was, in the end, not your best choice after all. Yet based on available data, it still *was* your best choice.

Now suppose that, *for some reason*, you decided that morning that a different path would be a better choice. Perhaps a cat crossed your usual path, and being superstitious, you decided to change your route. This is not logical: it is a decision based on a faulty algorithm. Yet it saved your life. Whether or not your actions were algorithmically determined, the end result is that a seemingly sub-optimal choice lead to a superior outcome.

Now suppose that true randomness is introduced: suppose your choice of which path to take to work is decided by the decay of a radioactive source. On the day in question, *for no algorithmic reason*, you decide that an alternate path would be a better choice. You have made a superior choice, but not based on the outcome of a deterministic algorithm.

This is just luck! Radioactive decay of a sample in Switzerland has little to do with the steering system of a downtown bus in New York. But this is, in part, because the critical event – the mechanical failure of the bus – is also effectively random.

12.4   When *Does* Free Will Make Sense?

Suppose the bus incident is not random, but an attempt to deliberately injure you. If you always follow the same path, and are unaware of this planned attempt, then you will be injured, despite making the best possible decision given all available/usable data. But if you use a random event to cause you to take a different path, that randomness could prevent the injury.

Now support (for some reason) that such attacks are likely. An algorithm that includes a degree of randomness may be your best choice, *and may be superior to any purely deterministic algorithm*. Certainly, changing your route every 3rd day won't yield a significant improvement in outcome, assuming the bus driver observes your habits and can determine your algorithm. Even a pseudo-random algorithm could be cracked. But an algorithm that incorporates true randomness – one whose outcome cannot be pre-determined – may yield a superior outcome.

12.5   Application to Ecosystem

So here's an idea for applying these concepts to EEXIST. Going back to the virtual ecosystem, we introduce two twists:

1. The system introduces a random element in the form of poisonous food: energy pellets that have negative energy. This food is identical in appearance to the regular food, there is no aspect of it that can be sensed by the organisms, except possibly after they've ingested it. Therefore, no matter how good their programming, they cannot take into account the possibility of poisonous food.
2. In some cases, individuals are allowed to make random decisions. There is a small chance that, despite their programming, their chemical balances will be slightly

perturbed once in a while, thus causing their behavior to be different from what would otherwise be their programmed (and, presumably, optimal) behavior.

A question then is, could it be that individuals endowed with a free will capability would somehow develop or exhibit survival advantages over the individuals who only act according to their strict programming?

To be honest, I didn't expect the poison food to have an impact in any profound way; yet my right-brain thought process was, *"what if* **intuition** *is somehow related to randomness?"* In other words, could some individuals develop a sense, a feeling like "something is wrong with that food" and avoid the poison, for no logical reason, no reason that can be predicted by its programming? This was an aspect I did not expect to see (and, in fact, does not seem to have been exhibited), but led me to the idea of introducing randomness into the behavior of EEXIST.

12.6 Experimental Setup

There are two principle changes to the system for these experiments:

- the occasional random adjustment of chemicals in individuals with the capacity for free will; and
- the appearance of poison in the food supply.

12.6.1 Free Will

Sequential numeric IDs are assigned to individuals at birth. To give nominally 10% of the population the capacity for free will, only individuals with IDs ending in 9 (henceforth referred to as "free-will individuals," or FWI for short) are candidates for free will choices.

Each FWI is seeded with a random time-till-choice (an integer from 0 to 63). Each time the system updates the state of a FWI (i.e. when it is about to step the EEXIST simulation), it first decrements the time-till-choice variable. Until this reaches 0, it has no effect. But once time-till-choice reaches 0, there is a possibility of a free will decision being made. A new random value V is generated (0-255), and if it is ≤ 48, the individual is considered to be making a free will decision. At this point, a random chemical level is selected (0-40), and injected into the individual's address space in the region between $x = 32$ and $x = 36$. If $V > 48$ then no free will is exercised at this step.

In either case, the time-till-choice is re-seeded with a new random value (0-63), and no possibility for a free will choice will occur until that new counter again reaches 0.

After this pre-processing, the system is ticked as usual.

12.6.2 Poison

Food is normally created pseudo-randomly, with an energy value between 20 and 40. In this modified system, each time food is created, there is a roughly 1/24 chance of it being poison. In this case, the energy value is multiplied by -5, resulting in a value between -100 and -200.

Recall that the energy of an organism is increased by the energy level of any food it eats. Similarly, poison would decrease an organism's energy level (by approximately $5\times$ as much as the typical energy benefit of normal food).

This is potentially a powerful poison: a value of -200 means it would immediately kill any organism with a current energy level of 200 or lower.

12.6.3 Sources of Randomness

Pesudo-random variables are often "good enough" in place of true randomness, but for these experiments (which felt pretty far out of the box to begin with), I decided real randomness was called for. If nothing else, this should eliminate the possibility of an organism "learning" the pseudo-random algorithm and thus being able to predict another organisms so-called free-will choices. Again, I did not expect EEXIST to be able to learn such things, but the *possibility* of a system learning to predict pseudo-random events was a good excuse to play with true randomness.

True randomness can be tricky to implement on a computer, since computers (by design) perform their operations according to a prescribed set of instructions, and their behavior should always be repeatable. Using something such as a current timestamp or system load can be used to generate pseudo-random data, but of course this is not truly random, and in theory, a powerful enough learning system might be able to predict the outcome of such generators.

The world of quantum mechanics provides a source of what is often considered "true randomness." Radioactive decay and thermal noise are two commonly used sources for the generation of true random data.

Fortunately, there are online sources for such data. One such source is HotBits (https://www.fourmilab.ch/hotbits/), which is operated by John Walker, Fourmilab, Switzerland. HotBits work by monitoring the radioactive decay of Caesium-137, and using the timing of that decay to generate random bytes. They provide an API for accessing this data through java (https://www.fourmilab.ch/hotbits/source/randomX/randomX.html).

As the capacity of HotBits to serve up random data is limited, one needs an API key when requesting data from the server. This key is used to track ones usage across time, and there are caps on how much data one may access in a given time period.

The relatively slow simulation speed of the EEXIST simulator reduced the rate at which random data was needed, but I found myself flirting with bandwidth limits, and thus opted to employ a second random source: https://www.random.org/.

Random.org is a site that will generate various random objects on request, including passwords, lottery numbers and sequences of integers. This site uses atmospheric noise as a source of random data, monitoring that noise with a series of spatially-distributed radio receivers. Random.org offers a free service for generating sets of random bytes, available through an HTTP API. This service is also bandwidth limited.

Neither HotBots nor Random.org allowed enough (free) bandwidth for the full-speed ecosystem, hence a combination of these two sources was used. A server runs on the local machine, and serves up random bytes on request by the EEXIST simulator. The server requests blocks of random bytes from the two sources, alternating between 3 blocks from

Random.org and then one block from HotBits. This combination allowed the ecosystem simulation to run at its full speed, without exceeding bandwidth limitations from either source.

12.7   Experimental Results

I had little expectation of any particular results from these experiments. While the idea that "free will injection" would provide a survival advantage, I had no logic reason to believe that randomly perturbing chemical levels in an EEXIST system would do anything other than degrade performance.

The system was run with 10% of the initial population having the capacity for free will decisions. While running, a separate monitor process created and posted a status webpage every 10 seconds. This page contained an image of the ecosystem (as shown in Figure 12.1), as well a lines of information (see Listing 12.1) about the status of the system and the makeup of the population.
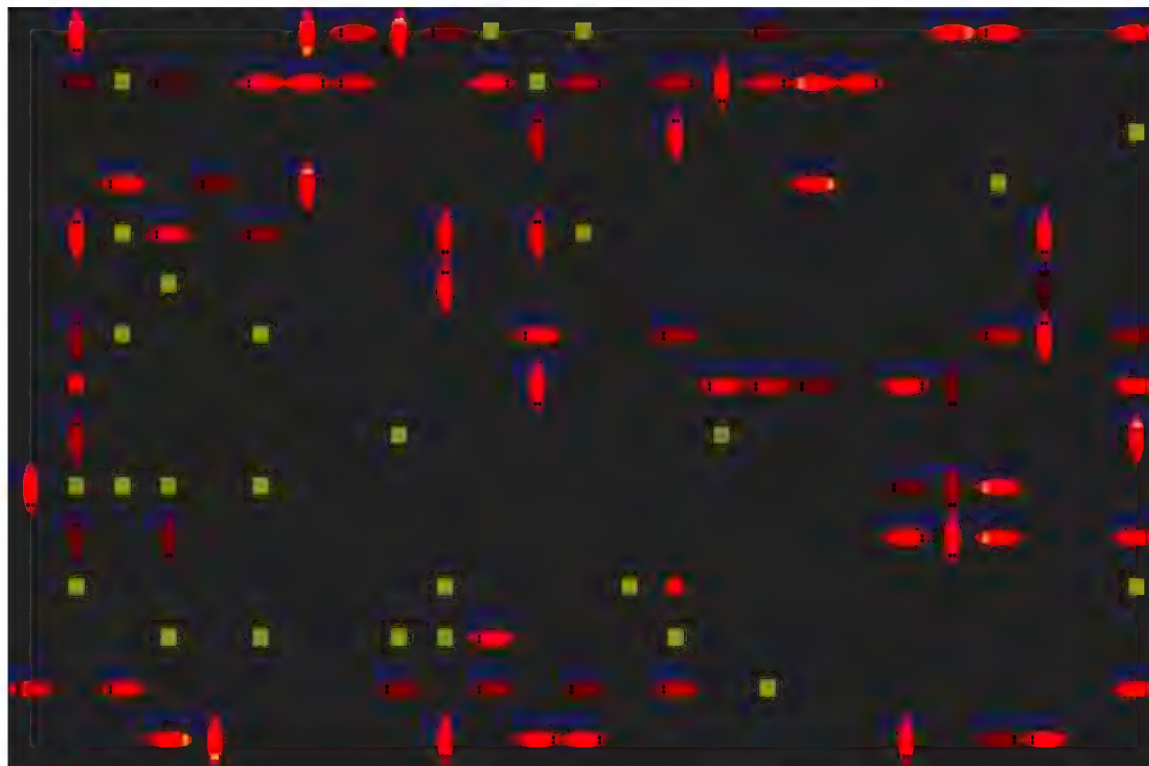
Figure 12.1: Sample Snapshot of Ecosystem Containing Free Will Individuals

Listing 12.1: Sample Ecosystem Status Report

```
Enhanced  ecosystem  with  free  will  in  certain  individuals
(random  bits  provided  by  random.org  and  Hotbits)
```

```
Tue  May  19  08:52:49  PDT  2020
Current  run:  0000000009
    PID  USER        % CPU % MEM      TIME+ COMMAND
22100  nick          68.8    1.4    3655:37  java
20739  nick           0.0    2.3  152:56.01  java
32417  nick           0.0    1.0    0:59.95  java


Image  from  2020-05-19  08:52:43.109646187
Instances  of  Free  Will
4772013:  Free  will:  individual  1179  injected  0.0
4772272:  Free  will:  individual  1659  injected  36.0
4772490:  Free  will:  individual  1339  injected  31.0
4772826:  Free  will:  individual  469  injected  27.0
4773077:  Free  will:  individual  1259  injected  18.0
4773134:  Free  will:  individual  1619  injected  36.0
4773280:  Free  will:  individual  1149  injected  39.0
4773328:  Free  will:  individual  879  injected  8.0
4773368:  Free  will:  individual  669  injected  17.0
4773805:  Free  will:  individual  629  injected  23.0
4774545:  Free  will:  individual  879  injected  35.0
4774793:  Free  will:  individual  629  injected  12.0
4775071:  Free  will:  individual  1189  injected  11.0
4775115:  Free  will:  individual  1019  injected  36.0
4775223:  Free  will:  individual  1189  injected  34.0
4775347:  Free  will:  individual  839  injected  23.0
4775484:  Free  will:  individual  849  injected  30.0
4775593:  Free  will:  individual  1649  injected  13.0
4776364:  Free  will:  individual  1249  injected  19.0
4776517:  Free  will:  individual  1629  injected  19.0
Recent  Deaths
4744482:  Creature  1616  is  dead
4745438:  Creature  1610  is  dead
4745766:  Creature  1631  is  dead
4747550:  Creature  1615  is  dead
4748411:  Creature  1640  is  dead
4749605:  Creature  1643  is  dead
4751075:  Creature  1621  is  dead
4754349:  Creature  1642  is  dead
4759461:  Creature  1626  is  dead
4760088:  Creature  1625  is  dead
4761822:  Creature  1647  is  dead
4766313:  Creature  1637  is  dead
4767902:  Creature  1641  is  dead
4768090:  Creature  1648  is  dead
4769498:  Creature  1653  is  dead
4770122:  Creature  1636  is  dead
```

```
4771809: Creature 1309 is dead
4772828: Creature 1660 is dead
4773486: Creature 1635 is dead
4773779: Creature 1645 is dead
New Births
4744484:New child (1646) from 641 and 1392
4745443:New child (1647) from 1479 and 1637
4745784:New child (1648) from 669 and 1361
4747551:New child (1649) from 1409 and 1647
4748480:New child (1650) from 1219 and 436
4749980:New child (1651) from 699 and 889
4751122:New child (1652) from 739 and 1601
4754385:New child (1653) from 1409 and 1647
4759466:New child (1654) from 16 and 1392
4760121:New child (1655) from 1269 and 564
4761839:New child (1656) from 739 and 883
4766314:New child (1657) from 1179 and 1652
4767940:New child (1658) from 1369 and 1656
4768091:New child (1659) from 1119 and 649
4769512:New child (1660) from 1649 and 1657
4770142:New child (1661) from 1119 and 1189
4771874:New child (1662) from 883 and 1629
4772841:New child (1663) from 16 and 1611
4773503:New child (1664) from 999 and 1663
4773781:New child (1665) from 1629 and 1119
Random Births
3675000: Old breeding: creatures 407 and 16 => 1200
3694000: Old breeding: creatures 629 and 629 => 1210
3750000: Old breeding: creatures 999 and 436 => 1226
3805000: Old breeding: creatures 879 and 883 => 1249
3864000: Old breeding: creatures 407 and 883 => 1268
3931000: Old breeding: creatures 699 and 829 => 1298
4011000: Old breeding: creatures 1119 and 889 => 1328
4029000: Old breeding: creatures 649 and 16 => 1334
4094000: Old breeding: creatures 469 and 1179 => 1355
4103000: Old breeding: creatures 163 and 309 => 1358
4108000: Old breeding: creatures 699 and 45 => 1361
4211000: Old breeding: creatures 1189 and 999 => 1392
4290000: Old breeding: creatures 309 and 641 => 1416
4393000: Old breeding: creatures 879 and 564 => 1465
4402000: Old breeding: creatures 629 and 1189 => 1472
4535000: Old breeding: creatures 564 and 939 => 1543
4563000: Old breeding: creatures 1369 and 1361 => 1556
4578000: Old breeding: creatures 889 and 1149 => 1562
4614000: Old breeding: creatures 91 and 740 => 1579
4734000: Old breeding: creatures 699 and 16 => 1638
```

```
Last recorded cycle: 4770001
Current Population
ID: AGE
45:4770001 (Gen 1)
16:4770001 (Gen 1)
91:4749001 (Gen 2)
163:4657001 (Gen 5)
195:4572001 (Gen 5)
245:4404001 (Gen 8)
407:3978001 (Gen 6)
429:3904001 (Gen 8)
436:3890001 (Gen 9)
469:3760615 (Gen 11)
564:3311001 (Gen 10)
629:3119086 (Gen 11)
641:3068001 (Gen 6)
649:3027408 (Gen 9)
669:2972490 (Gen 11)
672:2963001 (Gen 6)
699:2842906 (Gen 12)
729:2734606 (Gen 12)
739:2701804 (Gen 11)
740:2701001 (Gen 10)
839:2357187 (Gen 12)
849:2331446 (Gen 11)
879:2240214 (Gen 13)
883:2221001 (Gen 9)
889:2202131 (Gen 11)
939:2081001 (Gen 13)
999:1854947 (Gen 16)
1019:1759170 (Gen 12)
1119:1319033 (Gen 14)
1149:1255339 (Gen 12)
1179:1169743 (Gen 15)
1185:1149001 (Gen 13)
1189:1141604 (Gen 13)
1219:1044257 (Gen 14)
1249:965001 (Gen 14)
1259:944205 (Gen 19)
1269:899406 (Gen 12)
1309:800687 (Gen 17)
1339:723789 (Gen 15)
1349:687725 (Gen 13)
1361:662001 (Gen 13)
1369:644279 (Gen 21)
```

```
1392:559001 (Gen 17)
1409:498526 (Gen 11)
1429:447393 (Gen 19)
1479:345136 (Gen 19)
1509:286848 (Gen 20)
1539:245959 (Gen 22)
1556:207001 (Gen 22)
1601:109213 (Gen 17)
1611:95557 (Gen 16)
1619:71503 (Gen 7)
1622:63656 (Gen 12)
1623:57460 (Gen 13)
1624:56853 (Gen 14)
1629:47446 (Gen 15)
1630:45644 (Gen 13)
1632:44985 (Gen 23)
1635:39617 (Gen 15)
1636:39379 (Gen 15)
1638:36001 (Gen 13)
1639:35552 (Gen 12)
1644:29133 (Gen 14)
1645:27591 (Gen 12)
1646:25517 (Gen 18)
1649:22450 (Gen 24)
1650:21521 (Gen 15)
1651:20021 (Gen 13)
1652:18879 (Gen 18)
1654:10535 (Gen 18)
1655:9880 (Gen 13)
1656:8162 (Gen 12)
1657:3687 (Gen 19)
1658:2061 (Gen 22)
1659:1910 (Gen 15)
1660:489 (Gen 25)
```

General Website: https://SonglineSystems.com

This snapshot was taken after 4.77 million timesteps of the system. The most intriguing part is the last section ("Current Population"), which lists each individual in the current population, sorted by decreasing age, in the format:

ID:age (Generation number).

The first three entries are newborns (generation 1 or 2), and the next 4 entries are older (up to generation 8). These 7 individuals have IDs ending in 0-8, and hence are not free will individuals.

Beginning with the 8th entry, we begin to see free will individuals (having IDs ending in 9: 429, 469, 629, 649, 669, 699, and so on). And in this older population, this are

significantly more of the FWIs than the original 10%.

This was fairly startling to observe! Why would FWIs become so prominent in the society? IDs are assigned sequentially, so free will is not an inheritable property, so if by random chance a number of FWIs survived beyond the expected lifespan, their capacity for free will would not be directly propagated to future generations.

Listing 12.2 shows the same Current Population data at 5.94 million cycles. Here, the FWIs have claimed the top 10 positions (ranked by decreasing age) in the population list, as well as 18 of the top 20.

Listing 12.2: Population Makeup after 5.94 Million Cycles

```
Last recorded cycle: 5940001
Current Population
ID: AGE
649:4197408 (Gen 9)
729:3904606 (Gen 12)
879:3410214 (Gen 13)
1179:2339743 (Gen 15)
1259:2114205 (Gen 19)
1339:1893789 (Gen 15)
1869:810513 (Gen 16)
1909:742133 (Gen 24)
1949:671021 (Gen 17)
2129:488608 (Gen 23)
2162:463001 (Gen 23)
2259:409487 (Gen 31)
2309:379835 (Gen 32)
2317:374001 (Gen 18)
2359:347377 (Gen 26)
2369:342194 (Gen 22)
2489:266563 (Gen 32)
2559:229890 (Gen 27)
2569:224481 (Gen 17)
2629:189689 (Gen 24)
2645:182001 (Gen 14)
2669:172037 (Gen 28)
2693:159001 (Gen 17)
2697:158001 (Gen 24)
2729:143678 (Gen 32)
2806:111001 (Gen 17)
2809:110306 (Gen 25)
2840:97001 (Gen 16)
2870:83778 (Gen 33)
2897:73001 (Gen 17)
2917:66001 (Gen 22)
2949:51545 (Gen 25)
```

```
2974:41665 (Gen 33)
2978:41001 (Gen 20)
2988:37483 (Gen 31)
2995:35443 (Gen 24)
3007:31001 (Gen 25)
3009:30603 (Gen 34)
3013:29362 (Gen 32)
3018:28001 (Gen 16)
3030:25001 (Gen 25)
3038:21145 (Gen 26)
3043:20542 (Gen 31)
3045:19319 (Gen 25)
3047:19001 (Gen 24)
3049:18850 (Gen 27)
3050:18234 (Gen 29)
3056:15450 (Gen 17)
3057:14682 (Gen 25)
3061:13540 (Gen 18)
3062:12768 (Gen 25)
3065:11313 (Gen 33)
3067:9849 (Gen 33)
3068:9232 (Gen 32)
3069:8401 (Gen 33)
3073:7711 (Gen 34)
3074:7656 (Gen 18)
3075:6399 (Gen 34)
3077:5642 (Gen 29)
3080:5219 (Gen 27)
3081:5138 (Gen 23)
3083:4683 (Gen 24)
3085:3912 (Gen 34)
3086:3905 (Gen 28)
3087:3752 (Gen 25)
3088:3001 (Gen 17)
3089:2944 (Gen 32)
3090:2883 (Gen 21)
3092:2424 (Gen 19)
3093:2256 (Gen 35)
3094:2001 (Gen 20)
3095:1299 (Gen 25)
3096:1241 (Gen 25)
3097:774 (Gen 21)
3098:450 (Gen 33)
3099:1 (Gen 32)
```

At this point, it seemed clear that some sort of survival advantage was being exercised

by the free will individuals. This behavior persisted across multiple runs, and seemed somewhat immune to changes in karma. The system was enhanced to save more output, which was post-processed to try to understand this behavior. Unfortunately, no clear explanation has yet emerged.

The following figures show sample output from this post-processing. Figure 12.2 shows the change in population composition across the first 300,000 cycles. While the free will individuals begin as a minority, they quickly become the dominant part of the population.
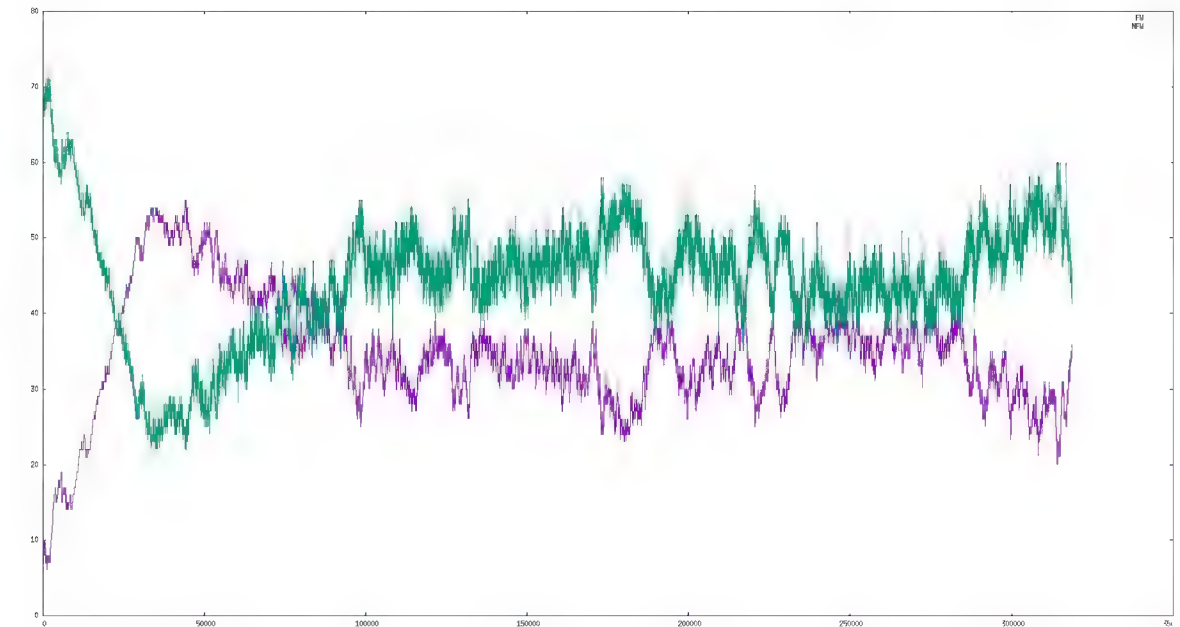
Figure 12.2: Population Composition, Free Will vs Non Free Will Individuals

Figures 12.3-12.5 presents the same information, but with the population restricted to only individuals whose age is at least 1000, 5000 or 10000 cycles.

This data seems to suggest that the dominance of FWIs is particularly significant in older populations. This makes some sense: 90% of all individuals are non free will individuals (NFWIs), and thus, depending on birthrates, one expects NFWIs to be a non-zero portion of the population. Restricting attention to older individuals factors out this aspect of population dynamics.

Continuing this consideration of age, Figure 12.6 shows the average of the population, with separate averages calculated for FWI and NFWI sub-populations. Not surprisingly, the FWIs appear to live longer (nominally $2\times$ to $10\times$ longer) than the NFWIs.

12.8   Conclusions/Next Steps

Logically, there are only a few reasons why one individual survives longer than another. Since all that is required to survive is not running out of energy, survival is a matter of
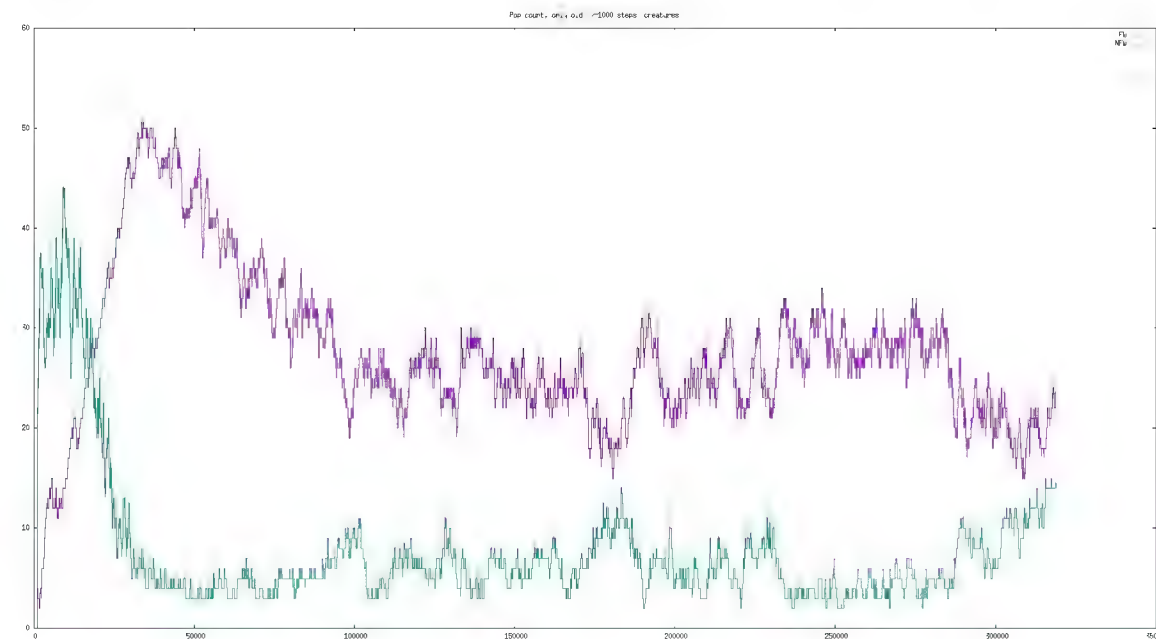
Figure 12.3: Population Composition (Individuals with Age>1000)
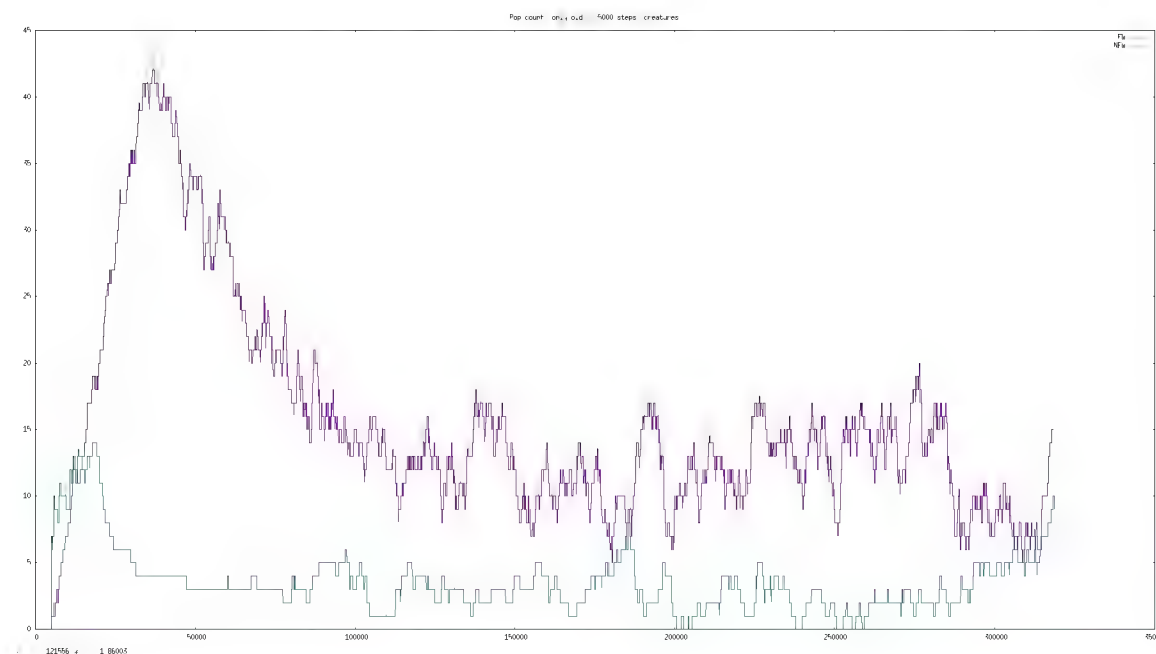
Figure 12.4: Population Composition (Individuals with Age>5000)

collecting more energy than is lost.

Energy is gained by the following actions:
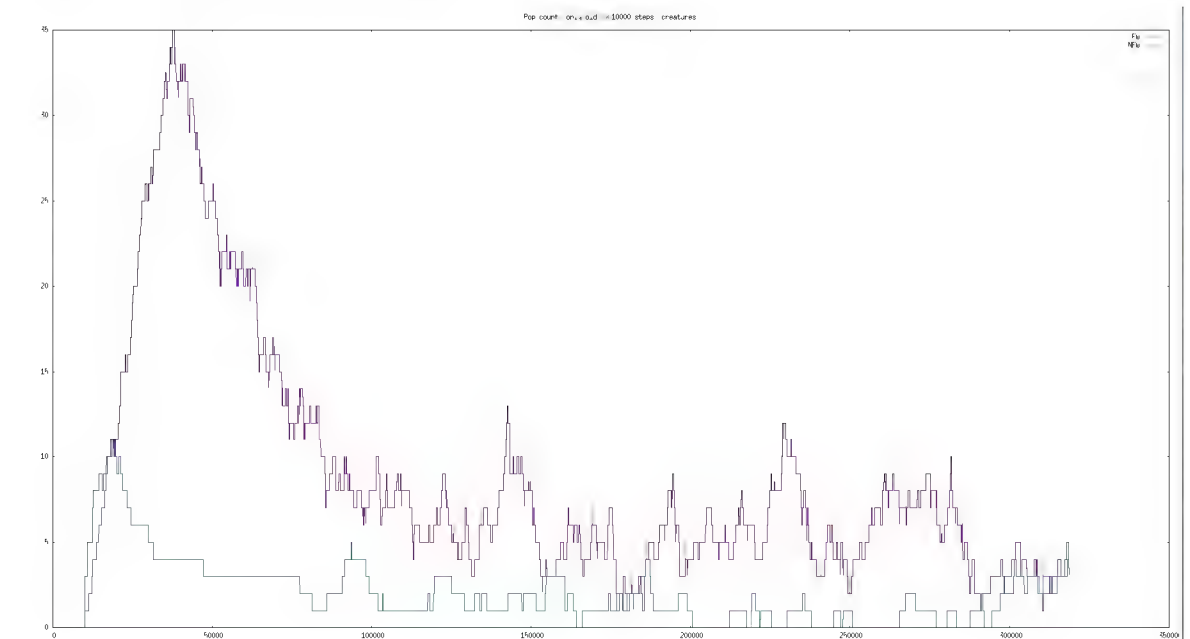
Figure 12.5: Population Composition (Individuals with Age>10000)
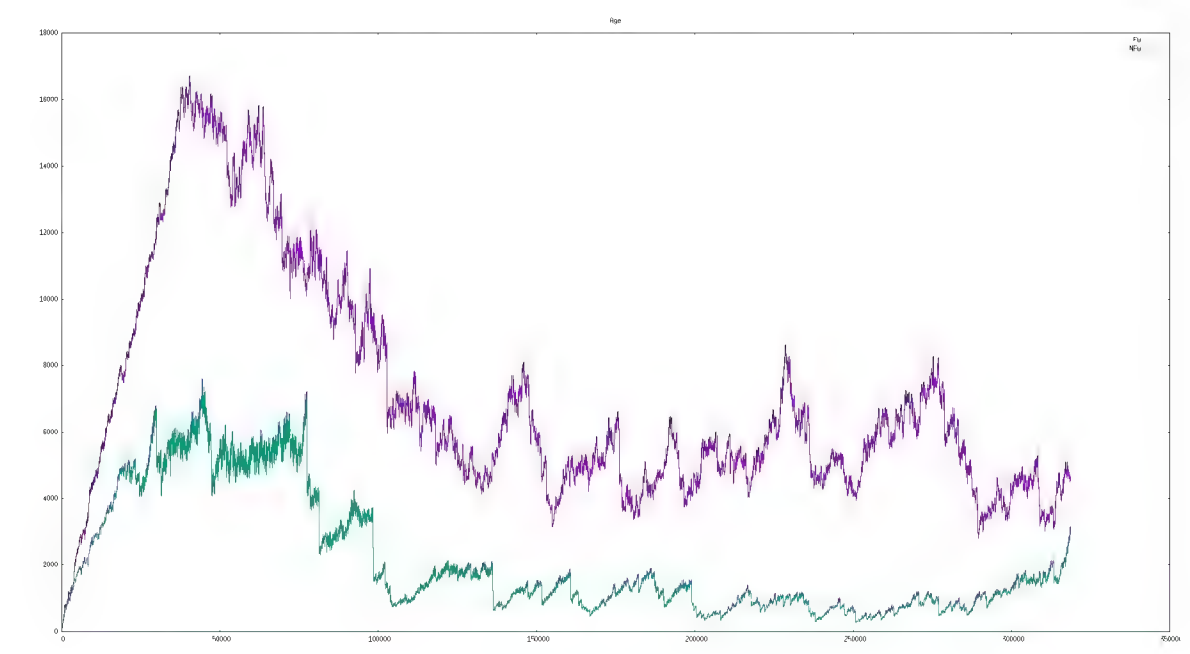
Figure 12.6: Average Population Age: Free Will vs. Non Free Will Sub-Populations

- attacking others
- eating (non-poisonous) food

Energy is lost by the following actions:

- any movement
- being attacked
- eating poison

So survival can best be ensured by some combination of the following goals:

- being better at attacking other;
- finding more food;
- evading attacks by others;
- avoiding poison; and
- avoiding movement that does not support the above goals.

Despite these observations and much analysis on the data produced by these runs, there has not yet emerged a clear explanation for the survival advantage of free will individuals. Figure 12.7 shows one such analysis result. This graph shows smoothed curves of population composition (% of population that has free will) and % of attacks made by FWIs against other FWIs. After somewhere around 60000 cycles, the FWI population is around 50% of the entire population, yet these individuals are attacking more NFWIs than FWIs.

Since an individual has no access to another individual's ID, it has no deterministic way of knowing whether another individual has free will or not. There is thus no reason to believe a FWI is more likely to attack a NFWI than another FWI. This suggests FWIs may be better at avoiding or escaping from attacks than NFWIs.
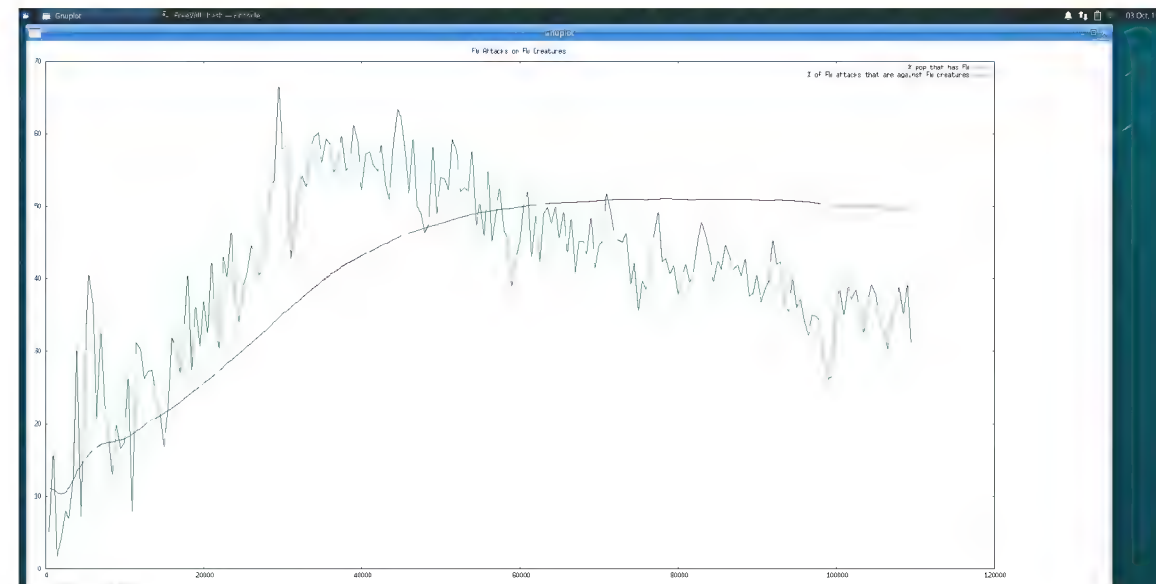
Figure 12.7: % of Population Having Free Will and % of FW Attacks on FWIs

Further analysis of population behavior and why FWIs show a survival advantage over NFWIs is ongoing.

12.8.1 Emotion Injection

As a possible continuation of this work, it might be interesting to add an *emotional* component to the operation of individuals controlled by EEXIST. This could take the form of an external system that monitors certain conditions and injects chemicals in response. For example, if the system's chemical levels are not changing much, this could be construed as "boredom," and a particular chemical infusion could result. If a repeating pattern is detected, this could be construed as "satisfaction," and a different chemical infusion could result. Failure to find a pattern might be "agitation," and so on. Even with no particular expectation of understanding or emulating actual emotions (whatever those might be!), it might still be interesting to see how these actions affect the system. This would be done in an ecosystem with some other metric besides simple survival.

12.8.2 Artistic Creation and Response

A related idea is to award merit to individuals based on some response they elicit form the population. For example, each individual could be given an output region that translates into, say, an audio signal; and a corresponding input region that receives audio input and translates it into a change in chemical levels in some region of its address space. Each individual would "perform" for the population, generating its output while the population responds to it. The scope of the effect could be measured externally, and an impact metric could be calculated based on the effect across the population. The individuals with the greatest effect (or, perhaps, with the most effect of a desired type) would be survivors in the genetic algorithm.

One can envision many such experiments.

IV Part Four - Next Steps

13. Next Steps

The experiments presented in this text represent a single thread of exploration through the space of possible experiments. The genetic approach has been used merely as a stop gap, since the process of designing algorithms for EEXIST is still poorly understood. Many of the design decisions that have been made (e.g. values for Δx, clipping of chemical levels, etc.) are likely enhancing or restricting the richness of the system. As almost all systems explored have varied from each other only in the equations of 10 linear bias gradients, the design space is certainly much larger than what has been explored so far.

While the GA work has yielded some insights into the capabilities of EEXIST, they do not illuminate the *bounds* of the architecture. In particular, only systems that produce very simplified behavior (which score well on the chosen metrics) have been studied.

Long-term goals are difficult to speculate on, other than the generic notion of "understanding the nature of the system's behavior" and looking for natural systems that EEXIST somehow emulates. Shorter-term goals are easier to enumerate, and are described below. This is not an exhaustive collection; it's simply a growing list of questions and ideas that have occurred throughout this work.

13.1 Evolving vs Learning

While the language of evolution and genetic algorithms has been used in these experiments, it's not clear that the development process is actually "genetic." The underlying structure

of each individual is identical: all that is changing is the setting of the bias at each location. This is not a fundamental change to e.g. the morphology of the individual; it seems more like a change to the "wiring" (since setting the bias basically changes the center point of an instruction's address space).

Moreover, while a population of individuals has been used to try out different such wirings, there's no reason these variations can't be explored within a single individual. Hence, it may be that the mechanisms being explored in these experiments are more akin to *learning* than to evolution. What is needed is a way for a single individual to retain the results of past experiments, and modulate their wiring over time. While this is somewhat a question of viewpoint, a change from working explicitly with a population of individuals to running all the mechanics inside a single individual likely has consequences for both efficiency and flexibility.

13.2   Input and Output

The basic model for input and output is only a first attempt at allowing interactions between EEXIST and the outside world. While the current model uses average SRC and DST levels inside a region, there are other possibilities:

- for output, one may require that *all* chemical levels inside a region be above or below a threshold (as opposed to the average value);
- for output, one may compare the difference between SRC and DST to a threshold;
- for input, one might inject chemicals *only once* rather than continually adding or removing chemicals to maintain the desired level.

13.3   Necessity of Bias

Adding bias gradients as the main genome variable seemed to be a key to successful evolution. However, there were <F2> other changes made alongside the switch to bias gradients:

1. the system is reset (i.e. initial chemical levels are restored) before each test, as opposed to leaving the system in its state following prior tests; and
2. experiments switched to testing systems on the entire set of possible inputs, as opposed to (for example) using random tests.

It would be interesting to re-visit evolving using pure chemical levels vs. bias gradients, following implementation of the above two changes. Not only might evolution still be feasible, it's unclear if bias actually adds any new capabilities to the system.

13.4   Uniqueness of Genomes

An open question is how a particular set of bias gradients translate into particular behavior. As a first step towards understanding this, an XOR gate was evolved 5 times, from 5 different initial (random) populations. After 37 generations, four of the populations had

evolved a perfect individual (12800/12800), while the best individual int he 5th population was performing at 99.91% perfection (12788/12800).

Figure 13.1 shows the bias gradients of these 5 individuals. As can be seen, there is no obvious similarity between these configurations.

(a)

(b)

(c)

(d)

(e)
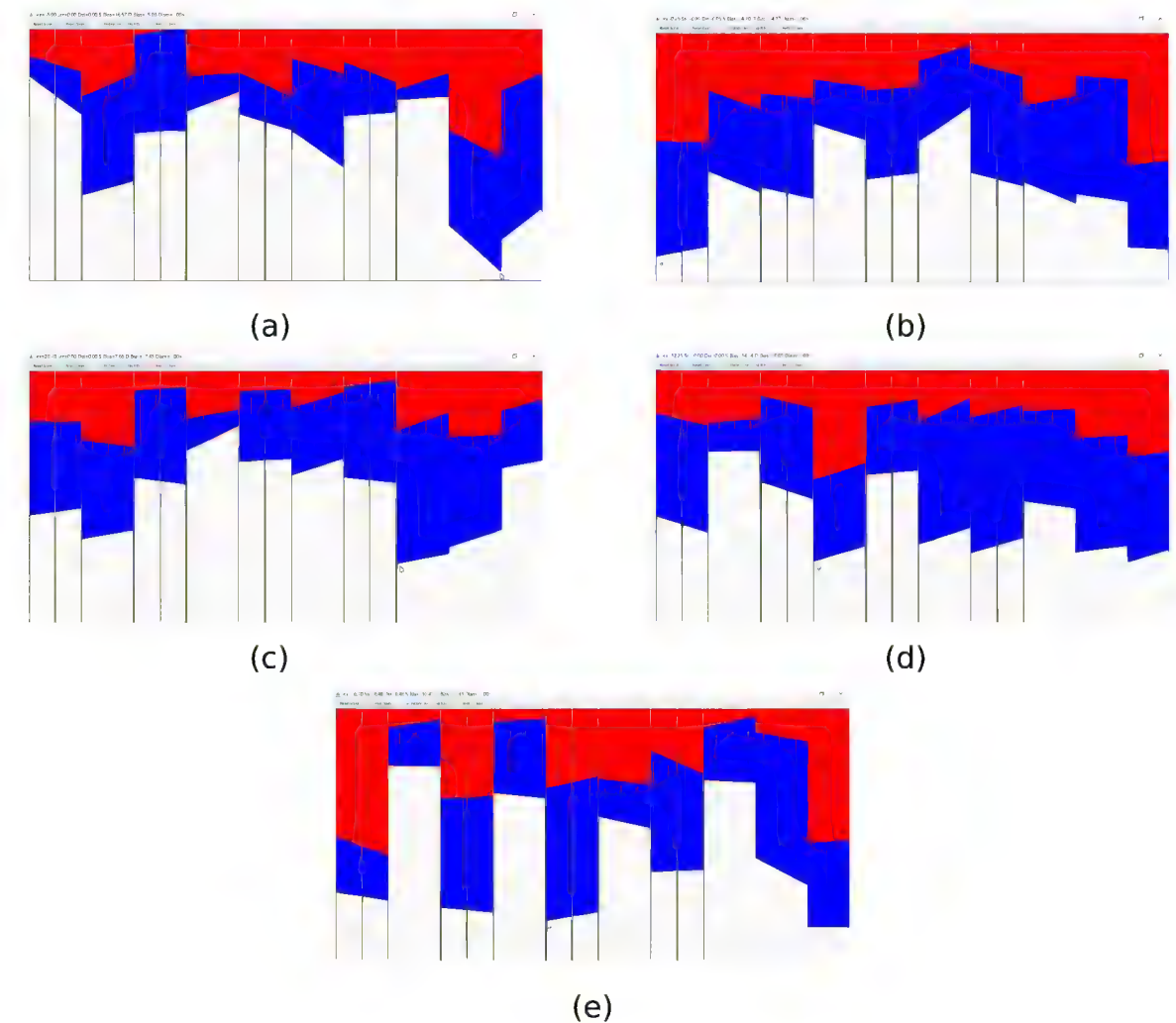
Figure 13.1: Bias Gradients for 5 XOR Configurations. (a)-(d) correspond to perfect individuals; (e) corresponds to an individual performing at 99.91% perfection.

Repeating this exercise with more-complex target behaviors might be more illuminating. At the least though, this exercise shows that the mapping between genomes and functions is definitely not injective. It would be interesting to repeat this over a very large number of runs (say 100 or 1000), and see if clusters of similar bias gradients appear.

13.5   Other Models and Implementations

The chemical model is a first idea about how to represent the enhanced ZBC. Other models are likely possible, and might expose different aspects of EEXIST which are being obscured by the SRC/DST chemical model.

Related to this is the question of implementation: how to build (at least theoretically) a physical version of EEXIST. The chemical model seems unlikely to be directly implementable: using chemical amounts to somehow address distant regions of chemicals seems unnatural. Such addressing in an electrical implementation can perhaps be more easily envisioned: voltage levels can be decreased along a resistive path, and their vanishing can be used to trigger a read or write at a remote location. Triggering at a voltage level near 0 can be used to introduce non-zero karma. The details of such a scheme are a topic of current research.

It is also possible that EEXIST already models some real-world system, and it is an ongoing quest to discover what such a system might be.

13.6   Other Areas to Explore

There are a number of other areas of EEXIST that can be explored:

- One can change the discretization (Δx and Δt) as well as the maximum and minimum values of x.
- The current work only sets chemical levels where $SRC = DST$: all variation is implemented via bias settings. There are many more possible configurations by allowing SRC and DST to be set independently.
- Most work has been done using *proportional flow* , where the rate at which chemicals move from a source location is proportional to the amount of chemicals present at that source. An alternative is *absolute flow* , where the flow rate is independent of the chemical levels (but still depends on diameter and karma). Recent experiments with absolute flow suggest that it works as well as proportional flow (depending on what the absolute flow rate is set to), which is useful because absolute flow may be easier to implement than proportional flow. More research is required into the differences between these two mechanisms.
- In the tic tac toe work, the address space was broken into 10 regions (to match the locations of the set of bias gradients in the genome). The system requires 9 inputs/outputs (a-i), each of which is allocated to one of the 10 regions as shown in figure 8.1); and the 10th region ($[36, 40]$) is used to inject initial chemicals to start the game (since EEXIST is moving first). Since all locations in the address space are used for input and output, it seems as if, in some sense, there's no room for computation: anytime chemicals are moved into most anywhere in the memory (except for $[36, 40]$), it contributes directly to a vote for a move in that square. While regions associated with moves that have already been made are in some sense "available," it feels somehow like there isn't enough space for pure computation. It would be interesting to either:

- – change the genome structure to use smaller bins, and change the input/output regions accordingly, to allow regions of the address space to be essentially unassigned; or
  - – increase the address space to cover, say, $[0, 80]$ instead of $[0, 40]$, while also increasing the number of regions in the genome to (in this case) 20.
- Figure 10.12 shows the death of individuals in a mixed population containing both trained and untrained individuals. It would be interesting to explore these dynamics in a population consisting entirely of trained (or untrained) individuals.
- The Gene and Genome class support other genetic structures beyond simple bias gradients. These still remain to be explored. Other modifications in the genetic mechanisms that can be explored include:
  - – the number of genes in the genome;
  - – alternative mating algorithms (choosing each gene from one parent or the other; selecting collections of genes from one parent or the other; taking a random mix of each parent's genes; and so on); and
  - – adjusting the mutation rate.
- Theoretically, one should be able to set negative diameters to effectively reverse the direction of chemical flow. This is relatively simple to explore, but hasn't been studied yet.
- karma is currently fixed for the entire system, across all time. Position-based karma, or karma which changes over time, are additional mechanisms that can be explored. Note that early genetic experiments tended to begin with small karma (for faster simulation), followed by increasing karma for better evolvability. More recent work (including the work reported in this text) has generally set $\kappa = 5$ for the duration of the runs.
- The address space is currently clipped at 0 and 40, but a wrap-around model may be more natural.
- The current flow model is based on pure flow from SRC to DST. An alternative is to view an instruction $SRC \rightarrow DST$ as simply connecting two locations, but not specifying a direction for fluid flow. Instead, chemicals flow between connected regions so as to move towards equilibrium, i.e. where the chemical levels at the SRC and DST locations would be the same. This somehow feels more natural, but is mostly unstudied.

13.7   Other Questions

Some questions about the system that remain unanswered:

- The simulation is based on a discretization of space and time. When the transition is made from finely-discretized to fully continuous, does the behavior change slightly, or is it possible that the behavior changes in some fundamental way?
- Another open question is how many (fundamentally) different behaviors are possible with a given genome structure. In the current set of experiments, the genome consists of 10 pairs of bias gradients, where a bias gradient is basically a pair of real numbers

between 0 and 40 (representing, say, the bias value at the start and end of a region). At some level, it seems like such a simple structure would be limited in its possible behaviors. However, if the system exhibits chaos , this limitation could be a non-issue, since even an infinitesimal change in one bias gradient might give significantly different behavior in the resulting system.

- If the instructions in the system are scaled by say $1/10$ (e.g. SRC and DST are each divided by 10); each instruction is relocated from location x to $x/10$; and the SRC and DST biases are scaled by $1/10$; κ is divided by 10; and the diameter of addresses in $(4, 8]$ are set to 0; then the resulting system should behave the same as the original, but will only occupy $1/10^{th}$ the space of the original system. Thus systems can be scaled (and of course the factor of 10 is not special: one could, in theory, scale the system to occupy one millionth of the original space). Similarly, adding an offset to the bias settings and relocating instructions by the same offset allows a system to be moved to different regions of the memory. Combining these, *it seems it's possible to include an infinite number of algorithms in a finite region of address space*. While perhaps limited in practical applications, the theoretical implications of this my be interesting.

Appendices

A. EEXIST Self Driving Car Algorithm - Caleb Hooper
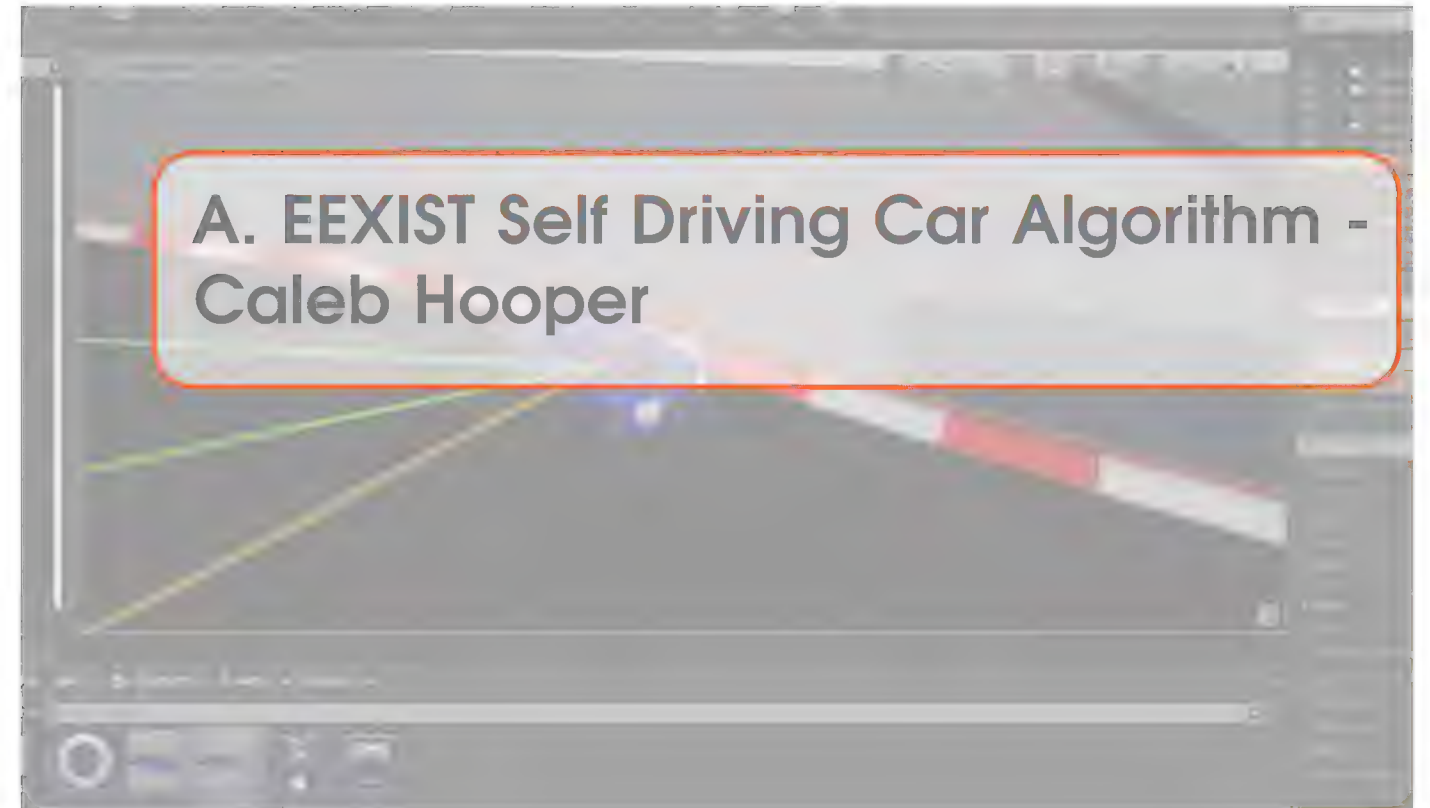
EEXIST Self Driving Car Algorithm

By: Caleb Hooper

Abstract:

The purpose of this project was to apply the EEXIST algorithm to a simulated virtual car to study the ability to control and drive complicated solutions and situations. With time the goal is to take data and experience and apply this to a real car for navigation or to be applied in video games or as an AI that controls a vehicle.

What Is EEXIST:

EEXIST is a form of computing created by Nicholas J. Macias which is an algorithm that can take input into a system process and return an answer that has been processed. How the input is processed is based on a number of rules and processes including.

- Collecting Data to process
- Running an algorithm on an array of data
- Selecting parts of this data as an output
- Training the system to process data correctly

This can be done into tested ways at the moment a 1D system or a 2D system. Essentially there is a grid of LENGTH and WIDTH and at each point on this grid (ie (1,1) (20,43)) there is a structure containing 4 floating point variables these variables represent 2 coordinates on the system itself this will be called the tube system and these coordinates will be used during a system update. Additionally, there is another set of data mimicking the shape and size of the system which contains 4 integers at each coordinate aka tube of the system. This data will be randomly generated at first but will be optimized through training this will be called the bias system. For the purposes of visualization, I will represent these 4 values as RGBA color and will be displayed as large square pixels in a window. Seen below you can see these two data banks the one on the left contains the current RGBA values of all coordinates in the tube system and the one on the right contains the RGBA values of the bias system.
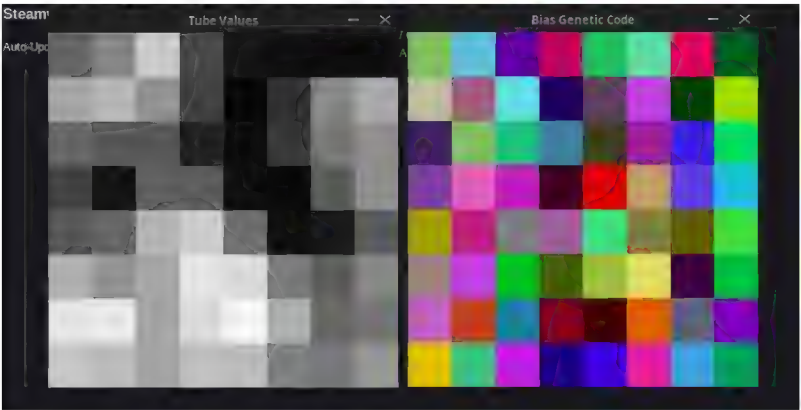
The process of system update is as followed.

1. Loop through each tube point in the system
2. Read two coordinates corresponding to the 4 floats stored in that tube
3. Move to the tube corresponding to the first two of those floats
4. Read the bias corresponding to the tube currently on
5. Add the bias value to the current coordinate and move to that corresponding tube
6. Remove a designated amount of value from that tube and the surrounding tubes
7. Repeat that process for the next 2 floating point values to find the destination
8. With the value place it into the destination calculated on the previous step
9. Read data prom predetermined tubes and treat these as answers to the given data
10. Apply this data to your problem in the case of this
11. Repeat for a designated amount of time for all tubes in the system
12. Once complete with a score representing the performance of the output rank the bias
13. When a suitable amount of biases have been testing take the best-performing biases to average them together and generate a new group of biases with a little bit of randomization mixed in
14. Rinse and repeat for as long as needed

Alternative Solutions:

Potentially better or different routes to go about solving this problem is by using neural networks because they provide a number of benefits. While EEXIST is much simpler to understand and implement it has a few downfalls

- It's not easily parallelizable
- Long training times

During training it's quite common for progression to follow a logarithmic curve and it can take quite a long time for bases to work there ways out of these slumps as seen below. The x-axis represents the current generation and the y-axis represents the performance of that generation.
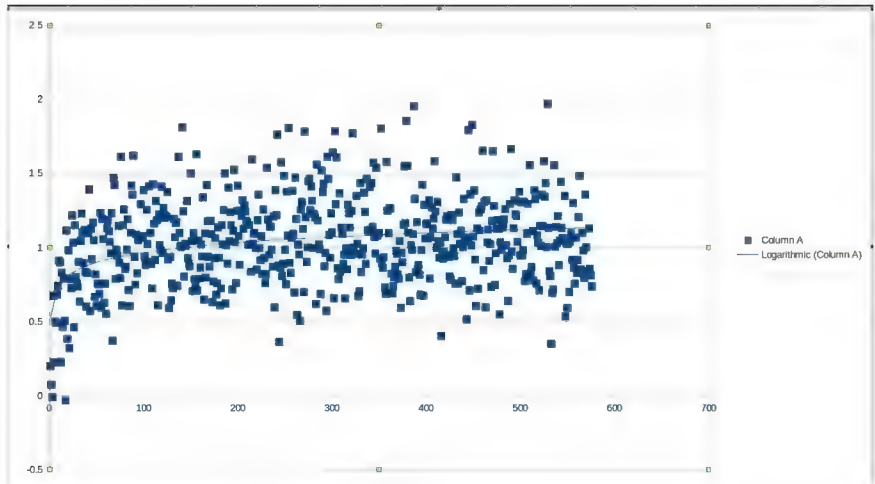
While these seem like crippling downfalls many implementation of this algorithm still have not been tested or explored and these problems could be fixed by different implementations.

Solution

Overview:

For the purposes of this project, the self-driving car will be completely simulated on the computer so that the focus can be around testing and studying the application of EEXIST on self-learning scenario with cars because of this, the project will be broken up into two programs and these programs will interact through reading and writing data to files. The following two programs will be discussed in detail below and can be done in any order.

Vehicle Simulation

Prerequisites:

- Unreal Engine 4.22
- IDE (Integrated Development Environment)
    - VS   (WINDOWS)
    - VS Code   (LINUX)
    - QT Creator   (LINUX)
- CUDA capable GPU (Optional)

To begin start by installing the Unreal Engine 4.22 The latest version may be used but this project has only been tested on version 4.22. Also an IDE will be needed for windows the best option is Visual Studios but for linux I will use VS Code. For getting into parallelization of the EEXIST algorithm a CUDA capable GPU will be needed.

Before beginning the IDE will need to be configured to work with unreal Engine. Note that this tutorial is meant to explain how this algorithm works and can be recreated not how to use unreal while the process will be explained some prior knowledge if the Unreal Engine may be helpful.

Project Setup:
       Once Unreal is successfully installed and the IDE is set up start by creating a new unreal blank project and create a blank C++ project.

Once you are in the editor you should be treated to a blank project like this.

The vehicle simulation will be broken into two main chunks the vehicle itself that is driven around and the track it drives on. We will start by creating the simulated vehicle. We will do this

by creating a new C++ class which can be done by selecting Add New->New C++ Class a window will show up and since we are creating something that can receive input from the player and EEXIST we will want to inherit from the pawn class.
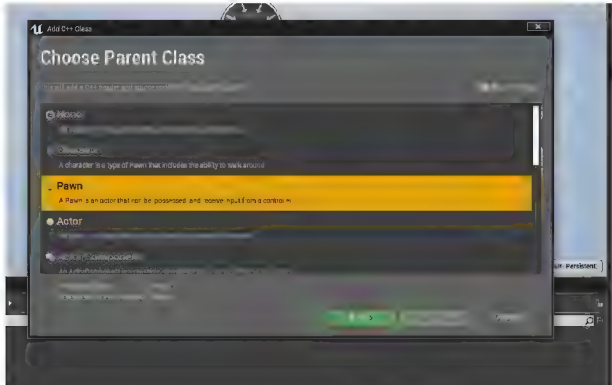
Click through the proceeding menus and name your class this will be our vehicle so I recommend something related to a car or vehicle. I will call mine MyVehicle and also make sure to mark your class as a public class before hitting create a class.

We will want a scene to start building our vehicle so we will create a new map. It's up to you how you set up your directory but in the content browser right click and create a new map.
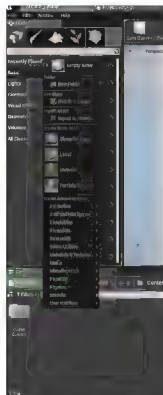
Select level and then double click on your new level to enter it for this I will just copy and paste the environment from the default level but feel free to decorate your level however you want. If you have lighting errors or any bad looking maps you can click on your directional light actors and set them to movable but I won't get into that.

Once you have an empty world place a platform into it that we will use to drive and test our vehicle on you can do this by selecting Geometry on the menu on the left and placing a box into the world. You can modify its properties like size and Position by adjusting the extent of the box in the details panel on the right side of your screen.
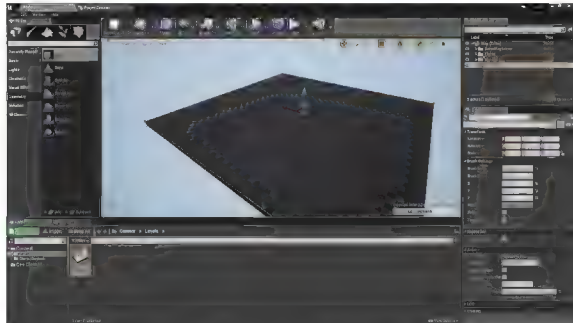
Now we can begin making the car we will need a few visual assets that represent the car you can make your own or used the ones provided at this download link.

https://drive.google.com/open?id=13tLQQxcP4lkbdqcWZepoG65cFs-xxj6w

It's up to you to figure out how to import and set up any extra visual effects for your Vehicle. If you used the provided one and imported it into unreal you should have a folder that looks like this.

Vehicle Code:
Now we will start programming our car so you will need to open up your IDE and navigate to the class we created earlier. The following parts will require a decent knowledge of the development process of games/simulations developed in unreal.

We will start by creating the following components for our vehicle class

- Collision
- Spring Arm
- Camera
- Skeletal Mesh

The following code will construct these components for us but you will need to be responsible for creating variables that are
`UPROPERTY(VisibleAnywhere, BlueprintReadOnly)`
in the header file. The code for creating these components is as followed.
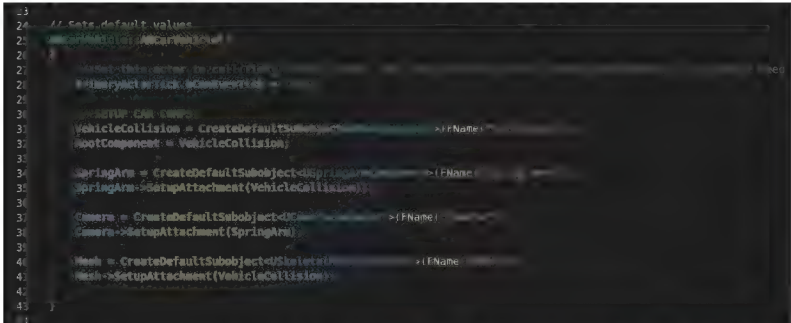
This will create the components needed for controlling viewing and using the car but we still need to set up and program how these will all interact together. To start we will want to store where the car starts on BeginPlay you can do that with the following code.

Next, we will want to program the update logic for the car this will be done in the tick function and will handle updating the position of the vehicle as the simulation plays out the following code is setup

That previous bit of code referenced a function called Update Position which is the actual function that simulates 1/60th of a second of movement and updates the car's position within the simulation.

Next, this is optional but if you wish to have a mode where you can control the car you will need to set up the input component which can be done with the following bit of code. Additionally, you will need to setup IO inside the Unreal settings which are fairly easy to figure out from a bit of research.

Next, we will set up the functions that are called when you trigger input into your system the following code is here which adjust the cameras viewing position and adjust the control settings for the car like throttle and steering.

This is basically all that's needed to control and drive your own car but now we need to set up the interfacing between this car simulation and the EEXIST AI simulation. This consists of 3 main functions.

The first of these functions is the collect data function. This function will be responsible for checking the current state of the car formatting this data and outputting to a file where the other EEXIST program can read it in. process it and return it to the Unreal program.

The next function is our input function this is responsible for taking data that was processed by our EEXIST program and applying it to our vehicle. It basically reads in data and sets the throttle and steering to the values inside the file.

Finally we have the reset function this will be responsible for resetting the car to its starting position and resetting cars throttle and steering.

This concludes the overview/setup of the vehicle simulation assuming you implemented these functions correctly you should now have a car that can drive around. Alternatively, the code is all available on the GitHub link below.

https://github.com/Vopraan/EEXIST-Vehicle

Next, we will develop the randomly generating track. We will start by creating a new C++ class inheriting the actor class instead of the pawn which is done as described earlier. Additionally, we will need a 3D model to represent the track in the simulation we will be using a premade one which can be downloaded here.

https://drive.google.com/open?id=1s6rfGRfzjG2l3M1C9_U-VqlWVqh0o8hV

The track will consist of two main parts a SplineComponent and an array of SplinemeshComponents that will line up and follow the spline Component. By doing this we can randomize position vectors to make a randomly generating track.

First, we will want to construct these components which will be done in our tracks constructor function which should look like the following

Since we want our track to generate after being spawned into the world we will want to invoke our generation which can be done with the OnConstruction function provided by unreal.

We will now write the function that handles the generation of roads and this will work by looping through a specified number of road segments and creating a Spline mesh at each one and calculating starting and ending states for that segment of road.

Optionally we will include a function that spawns random debris on the road that the vehicle could potentially navigate around as an extra obstacle although we will not be using this in our testing.
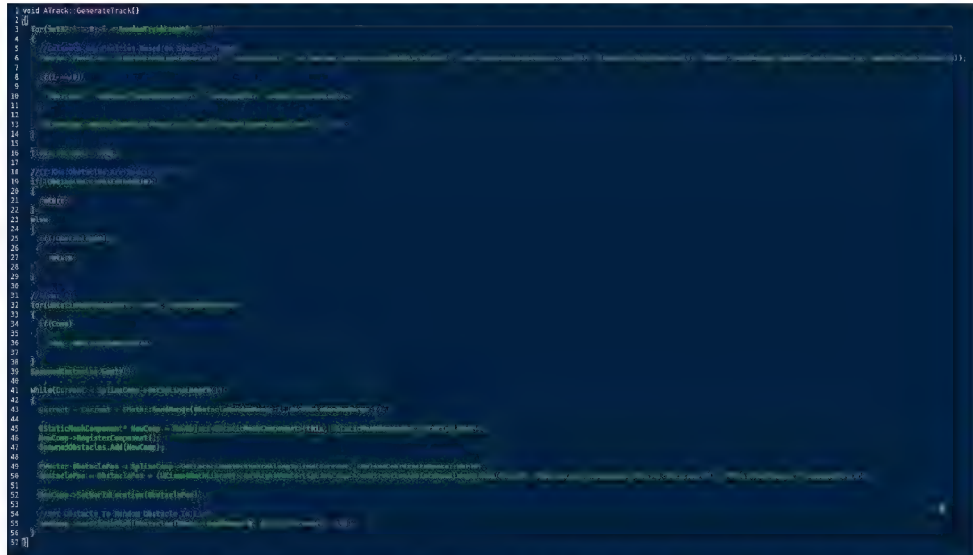
That concludes our track setup and vehicle simulation setup depending on how you implemented it you should have a vehicle that can drive around on a track or just sit there and do nothing waiting for input from the EEXIST program. Source code is available on GitHub.

https://github.com/Vopraan/EEXIST-Vehicle
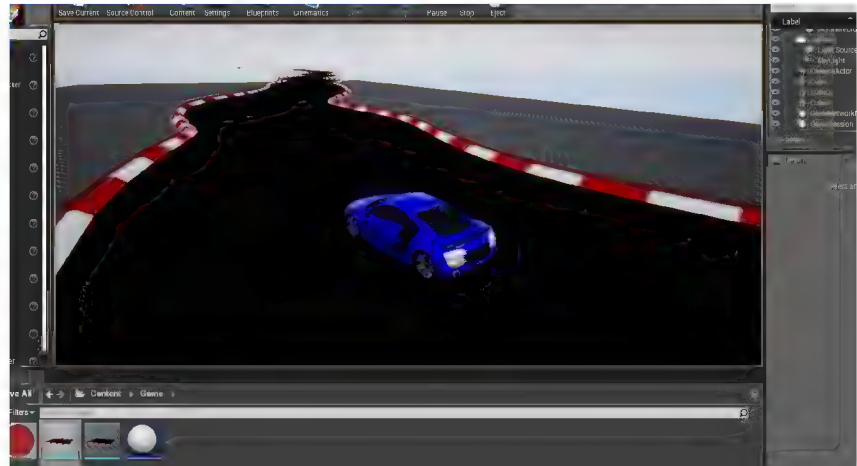
EEXIST Algorithm

Prerequisites:

Before beginning on creating the EEXIST algorithm a few libraries and programs will be required.

- SDL-2.0 (Windowing Manager)
- IDE (Integrated Development Environment)
  - VS   (WINDOWS)
  - VS Code   (LINUX)
  - QT Creator   (LINUX)
- CUDA 9.0 or Later (Optional)

It will be up to you to install your IDE of choice but if you are on a Debian based build of Linux the SDL library can be installed with the following terminal command.

```
sudo apt-get install libsdl2-2.0
```

CUDA may be installed by using

```
sudo apt-get install nvidia-cuda-toolkit
```

Once SDL library is installed you can begin creating the EEXIST algorithm. Note that this is a separate program from the vehicle simulation so would be created in a separate project for your IDE of choice.

Before we start creating the AI we will want a way of visualizing how the AI works which will be done by creating two windows that display the current state of the EEXIST system and the current Bias being used by the system. To do this I will create a class and I will call mine Window.cpp. We will want to do basic SDL initialization in the constructor and handle creating the window which can be done with the following code.
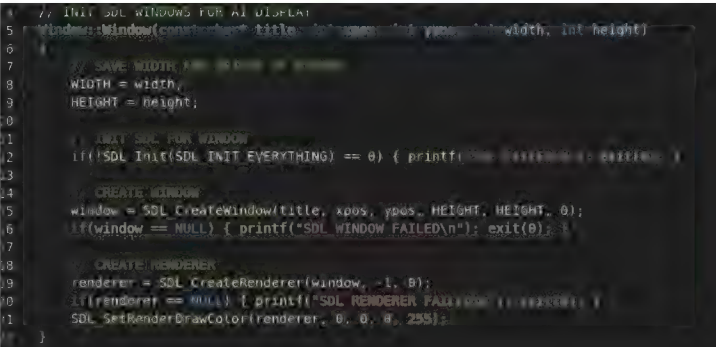
We will want to update the display and we will defer this to an update function which will take a reference to a system and print its data to the window by means of color RGBA values come from either the chemical values of each tube or the bias values for that tube depending on the window being displayed.

That takes care of the window display but we will need to create two of these classes inside of the main function and set up a while loop that runs for the duration of the program.

This will only need to run while the program has not been exited by clicking the x button so we will need a function that checks for this.

Finally, we update for the AI needs to be called every time through the loop which will handle resetting loading in data and updating the System with the EEXIST algorithm. Before we do that we will want to create a new class to handle all of the training/learning.

If you plan to experiment with CUDA and multithreading for this algorithm you will want to allocate space for your system and its bias which will be 4 floats for each coordinate for the system array and 4 ints for each bias for that system. For multithreading depending on how you do it you will want a buffer which will be an array of these systems that was just described the array should contain WIDTHxHEIGHT systems in it. For example, if you want a 50x50 system

you should have 50x50 array of tubes and 50x50 array of bias and a buffer which contains 50x50 systems of which was just described. This part is optional though and only needed for experimenting with CUDA multithreading on GPU. Note that this can be quite memory intensive and may require a higher end GPU with 2+ GB of memory.

Next, the update system function will direct its task depending on the state of EEXIST if we can the simulation is done we will find a file written to disk that says update and when this is found we will start the task of clearing and resetting EEXIST for the next test which includes saving the current bias if it is better than other biases saved for this generation. If it is not time to reset we will find a file containing data for the current update which is just what the car sees in the simulation which is a direction vector with magnitude pointing out from different rotations around the car which is basically what the EEXIST algorithm can see.
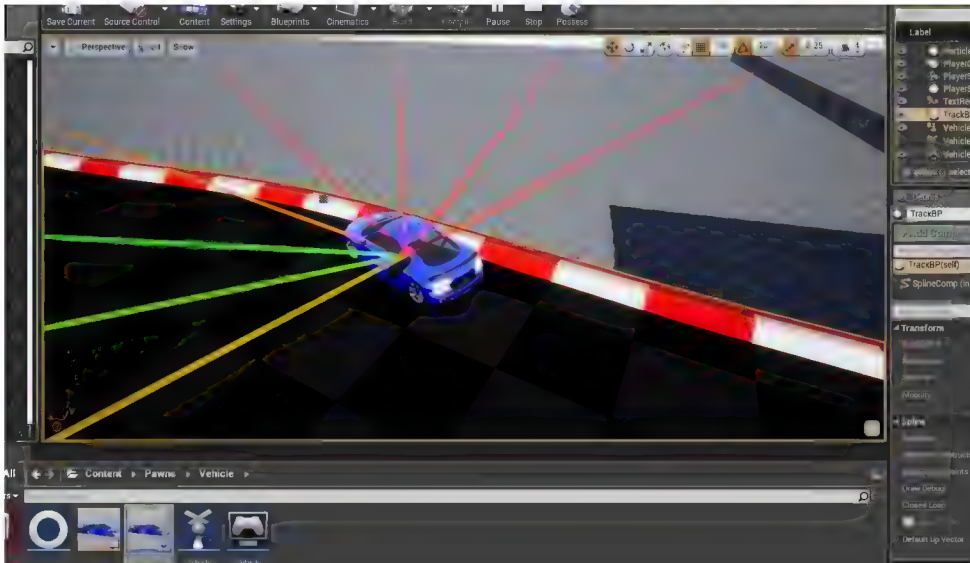
The reset process is the most complicated it will involve checking if a bias waiting to be tested for this generation still exists loading that in or if not generating a whole new generation of biases to test this can all be done with the following.

```
64    // PERFORM RESET BY GNERATION NEW GENRATION OR USING EIXTING GNERARION BIASES
65    void EEXIST::Reset()
66    {
67        // CHECK IF THERE
68        String BiasTest;
69        GetBiasToTest(&BiasTest);
70
71        // IF WE STILL HAVE BIAS LEFT LOAD THEM IN AND FINISH RESET
72        if(BiasTest.toChar() != NULL) { LoadInBias(&BiasTest); ResetTubes(); remove(NEW);
73
74        // IF THERE ARE NO BIASES
75        GenGeneration(); Reset();
76    }
```

```
64    // PERFORM RESET BY GNERATION NEW GENRATION OR USING EIXTING GNERARION BIASES
65    void EEXIST::Reset()
66    {
67        // CHECK IF THERE
68        String BiasTest;
69        GetBiasToTest(&BiasTest);
70
71        // IF WE STILL HAVE BIAS TO TEST LEFT LOAD THEM IN AND FINISH RESET
72        if(BiasTest.toChar() != NULL) { LoadInBias(&BiasTest); ResetTubes(); remove(NEW);     }
73
74        // IF THERE ARE NO BIASES TO TEST LEFT GENERATE A NEW GENERATION
75        GenGeneration(); Reset();
76    }
```

This function checks the disk for bias waiting to be tasted and returns the directory of the first one found.

```
123    // GET NEXT BIAS TO TEST RETURNS NULL IF NO BIASES LEFT
124    void EEXIST::GetBiasToTest(String* BiasFileDir)
125    {
126        int Check = 10;
127        if(BiasFileDir == NULL) { return; }
128
129        while(Check >= 1)
130        {
131            sprintf(BiasFileDir->toChar(), "%s%d", GEN, Check);
132
133            if(DoesFileExist(BiasFileDir)) { return; }
134
135            Check--;
136        }
137
138        BiasFileDir->empty();
139    }
140
```

This next function takes a given direction of a bias to test and loads that into the system.

```
141    // LOADS IN A GIVEN BIAS FILE INTO THE SYSTEM BIAS
142    void EEXIST::LoadInBias(String* BiasFile)
143    {
144        // VERIFY BIAS FILE EXIST
145        FILE* File = fopen(BiasFile->toChar());
146        if(File == NULL) { printf("COULD NOT ...                                  ");  }
147
148        // READ IN FILE DATA
149        for(int y = 0; y < SIZE; y++)
150        {
151            GetLine(SIM * SIZE, ...
152            ReadLine, (TS * SIZE) ...
153
154            // GET EACH LINE ...
155            ...
156        }
157
158        }
159
160        fclose(File);
161        remove(BiasFile->toChar());
162    }
```

This function resets the current state of all tubes in the system to their default state of the LENGTH of the system divided by 2.

```
286    void EEXIST::ResetTubes()
287    {
288        // INIT ALL TUBES TO DEFALT VALUE
289        for(int y = 0; y < SIZE; y++)
290        {
291            for(int x = 0; x < SIZE; x++)
292            {
293                Sys->Tubes[x][y].SrcX = SIZE / 2.0f;
294                Sys->Tubes[x][y].SrcY = SIZE / 2.0f;
295                Sys->Tubes[x][y].DesX = SIZE / 2.0f;
296                Sys->Tubes[x][y].DesY = SIZE / 2.0f;
297            }
298        }
299    }
300
```

If we don't find a bias to test we will want to generate a new generation of biases each based off the best performing bias/biases of the previous generation.

```
// GENERATE NEW GNERATION WITH BEST TWO BIASES FROM PREVIOUS GNERATION
void EEXIST::GenGeneration()
{
    String Breed[GENERATION_SAMPLE];

    // GET BIASES TO BREED
    for(int i = 0; i < GENERATION_SAMPLE; i++)
    {
        sprintf(Breed[i].toChar(), "%s%s%d", BIASES, "Bias", i + 1);
        if(!DoesFileExist(&Breed[i])) { GenRandomBias(&Breed[i]); }
```

```
bias Biases [GENERATION_SAMPLE] [SIZE][SIZE];
bias Avg                 [SIZE][SIZE] = { bias(0, 0, 0, 0) };

for(int i = 0; i < GENERATION_SAMPLE; i++)
{
    FILE* File = fopen(Breed[i].toChar(), "r");
    if(File == NULL) { printf("FAILED TO LOAD BIAS FILE : %d :
%s\n",i, Breed[i].toChar()); exit(0); }

    char EMPTY[2];
    fgets(EMPTY, 2, File);

    for(int y = 0; y < SIZE; y++)
    {
        string Absolute * SIZE = y;
        fgets(Line.toChar(), ((x * SIZE) + 2, File);

        for(int x = 0; x < SIZE; x++)
        {
            Line.toChar()[((x * 10) - 1] = '\0';
            sscanf(Line.toChar() + (x * (%d %d %d %d",
&Biases[i][x][y].SrcX, &Biases[i][x][y].SrcY, &Biases[i][x][y].DesX,
&Biases[i][x][y].DesY);

            Avg[x][y].SrcX = Avg[x][y].SrcX + Biases[i][x][y].SrcX;
            Avg[x][y].SrcY = Avg[x][y].SrcY + Biases[i][x][y].SrcY;
            Avg[x][y].DesX = Avg[x][y].DesX + Biases[i][x][y].DesX;
            Avg[x][y].DesY = Avg[x][y].DesY + Biases[i][x][y].DesY;
        }
    }

    fclose(File);
}

for(int x = 0; x < SIZE; ++y)
{
    for(int y = 0; y < SIZE; ++k)
    {
        Avg[x][y].SrcX = Avg[x][y].SrcX / GENERATION_SAMPLE;
        Avg[x][y].SrcY = Avg[x][y].SrcY / GENERATION_SAMPLE;
        Avg[x][y].DesX = Avg[x][y].DesX / GENERATION_SAMPLE;
        Avg[x][y].DesY = Avg[x][y].DesY / GENERATION_SAMPLE;
```

```
    }

    for(int i = 1; i <= GENERATION_SIZE; i++)
    {
        string FileName;
        sprintf(FileName.toChar(), "%d", GEN, i);
        FILE* File = fopen(FileName.toChar(), "w");
        if(File == NULL) { printf("FAILED TO BIAS FILE: %s\n",
FileName.toChar()); exit(0); }

        for(int y = 0; y < SIZE; y++)
        {
            for(int x = 0; x < SIZE; x++)
            {
                if(rand() % 100 < MUTATION) { fprintf(File, "%d %d
%d ", (rand() % SIZE), (rand() % SIZE), (rand() % SIZE), (rand() %
SIZE)); }

                else { fprintf(File, "%d %d %d ", Avg[x][y].SrcX,
Avg[x][y].SrcY, Avg[x][y].DesX, Avg[x][y].DesY); }
            }

            fprintf(File, "\n");
        }

        //fclose(File);
    }

    remove(GEN);

    for(int i = 0; i < GENERATION_SAMPLE; i++)
    {
        remove(Breed[i].toChar());
    }
}
```

```
// GENERATE A BIAS FILE IF DIRECTION WITH RANDOM BIAS VALUES
void EEXIST::GenRandomBias(String* mFile)
{
    // OPEN FILE
    FILE* File = fopen(mFile->toChar(), "w");
    if(File == NULL) { printf("FAILED TO MAKE NEW BIAS FILE : %s", mFile->toChar()); exit(0); }

    fprintf(File, "% %d\n");

    for (int y = 0; y < SIZE; y++)
    {
        for (int x = 0; x < SIZE; x++)
        {
            fprintf(File, "%3d %3d %3d %3d ", rand() % SIZE, rand() % SIZE, rand() % SIZE, rand() % SIZE);
        }
        fprintf(File, "\n");
    }

    fclose(File);
}
```

If we don't want to reset at all and a file is present on the disk with the vision data from the vehicle then we want to perform an update on our system. To do this we will load in data from file load in vision data into selected read-only tubes which by default here the first row is always read-only tubes meaning the value there given is set and can not be changed but the can still cause transfers to happen through the system but other tubes and themselves cannot cause transfers in or out of these read-only tubes. After the system has been updated it we will check specific output tubes for there value and write them to file to be read in by the simulation and plugged into the throttle and steering of the car.

```
// PREAPARE SYSTEM UPDATE
void EEXIST::PrepareUpdate()
{
    // MAKE SURE SAFE FILE EXIST BEFORE UPDATING
    String SafeFile(SAFE_OUTPUT);
    if(!DoesFileExist(&SafeFile)) { return; }

    // OPEN OUTPUT FROM UNREAL
    FILE* OutputFile = fopen(OUTPUT, "r");
    if(OutputFile == NULL) { return; }
    remove(SAFE_OUTPUT);

    // READ AND STORE NUMBER OF INPUT TUBES
    int OutTube = 0;
    char OutTubeString[16];
    fgets(OutTubeString, 16, OutputFile);
    sscanf(OutTubeString, "%d", &OutTube);

    // LOOP THROUGH INPUT TUBES LOAD IN VALUES
    for(int i = 0; i < OutTube; i++)
    {
        float R, G, B, A;
```

```
        char ValString[64];
        fgets(ValString, 64, OutputFile);
        sscanf(ValString, "%f %f %f %f", &R, &G, &B, &A);

        Sys->Tubes[i][0].SrcX = R;
        Sys->Tubes[i][0].SrcY = G;
        Sys->Tubes[i][0].DesX = B;
        Sys->Tubes[i][0].DesY = A;
    }

    // LOAD SCORE VALUE
    String OutScoreString(16);
    fgets(OutScoreString.toChar(), 16, OutputFile);
    sscanf(OutScoreString.toChar(), "%f", &Score);

    // CLOSE OUT FILE AND REMOVE
    fclose(OutputFile);
    remove(OUTPUT);

    // PERFROM DATA TRANSFERS ON SYS
    StartSysUpdate(Sys, Buffer);

    // PERFORM ADD ON GPU
    StartSysAdd(Sys, Buffer);

    // OUPUT NEW TUBE STATES TI FILE FOR CONTROLLING THE SIM
    FILE* InputFile = fopen(INPUT, "w");
    if(InputFile == NULL) { printf("FAILED CREATE INPUT FILE\n");
exit(0); }

    // OUPUT  ACCLEARATION TO FILE
    float Vall = 0;
    for (int y = SIZE - 1; y > SIZE - 1 - OUTPUT_REIGON; y--)
    {
        for (int x = 0; x < OUTPUT_REIGON; x++)
        {
            Vall = Vall + Sys->Tubes[x][y].SrcX + Sys->Tubes[x][y].SrcY +
Sys->Tubes[x][y].DesX + Sys->Tubes[x][y].DesY;
        }
    }
    Vall = Vall / (OUTPUT_REIGON * OUTPUT_REIGON * 4);
    Vall = Vall / (SIZE - 1);
    Vall = (Vall * 2) - 1;
    fprintf(InputFile, "%f\n", Vall);

    // OUPUT STEERING TO FILE
    float Vall = 0;
    for (int y = SIZE - 1; y > SIZE - 1 - OUTPUT_REIGON; y--)
```

```
            {
        for (int x = SIZE - 1; x > SIZE - 1 - OUTPUT_REIGON; x--)
            {
                Val2 = Val2 + Sys->Tubes[x][y].SrcX + Sys->Tubes[x][y].SrcY +
Sys->Tubes[x][y].DesX + Sys->Tubes[x][y].DesY;
            }
    }
    Val2 = Val2 / (OUTPUT_REIGON * OUTPUT_REIGON * 4);
    Val2 = Val2 / (SIZE -1);
    Val2 = (Val2 * 2) - 1;

    fprintf(InputFile, "%f\n", Val2);
    fclose(InputFile);

    // WRITE OUT SAFE FILE TO LET UNREAL WE ARE DONE MODIFYING THE INPUT
FILE
    fclose(fopen(SAFE_INPUT, "w"));

    // REPORT RECORED STEER AND ACCLE VALUES
    printf("ACCEL: %f STEER: %f SCORE: %f\n", Val1, Val2, Score);
}
```

Finally the actual algorithm that provides the system update we train on this will loop through each tube in the out system find a coordinate to take and put the value from and to then offset by the bias currently testing and move some value from one tube to the other including some values from tubes around it which is called the KARMA value. This will be done for every tube in the system and can become quite taxing for the computer on larger systems

After this step the AI should be created and when this program runs two windows should pop up one will be completely gray representing the default value of chemical in each of the systems tubes and the second representing the current bias waiting to be test if all is set up the test will begin as soon as the simulation begins and you will see biases cycle in and out of the window as it continues to test. Additionally, the tube values will display interesting fluid like pattern as you can see the EEXIST algorithm generate potential solutions to its given data set.
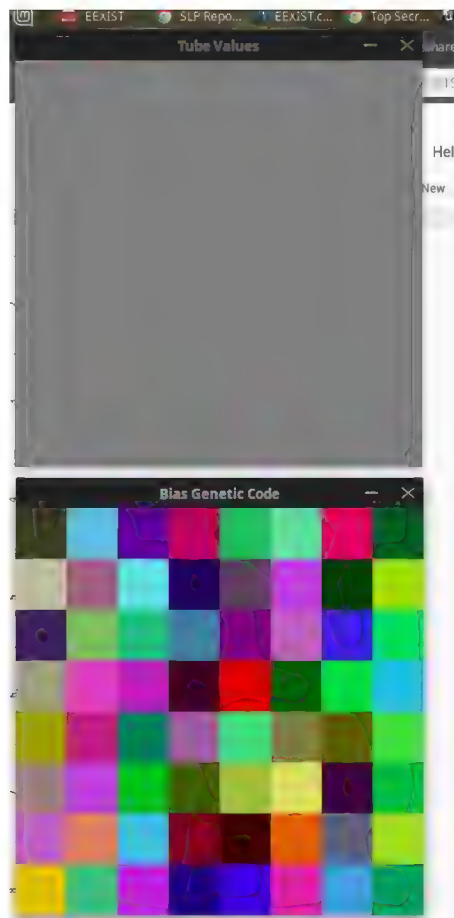
Source code is available on GitHub for this here.

Enhancements:

Currently the applications and capability of EEXIST is largely unknown and as of now the best improvements will come from experimentation and testing. Additionally if the goal is not to use EEXIST algorithm then neural networks are a good option to explore as they have been heavily explored and proven to give results.

Results:

Overall a car can drive down a track was achieved but not to the extent planned. After testing the best performing biases ended up being able to drive just over 100 meters without messing up but did so as a very slow pace and were not good enough to go replacing any AI anytime soon. More performance could be achieved by more training but this could take long amounts of time as improvements follow a logarithmic curve as seen below.

X = Generation   Y = Average Distance (Decimeters)
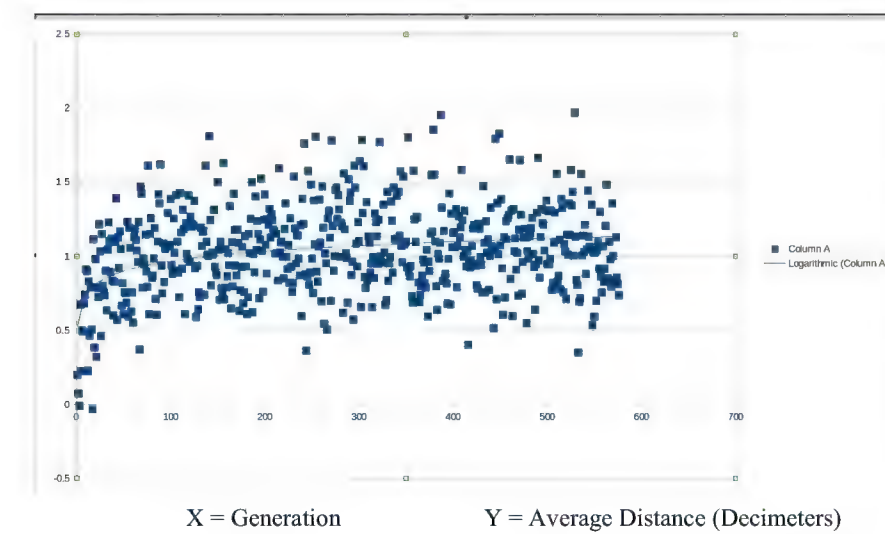
Most likely more testing and experimentation will be needed before this algorithm can actually possibly be applied to video games or vehicles.

Conclusion:

EEXIST won't be controlling real cars or driving the latest video games anytime soon but the algorithm shows promise and with more testing and additional implementations it could potentially drive the car as imagined as the algorithm has also been shown to work with facial recognition and various other small games where it was set to control an actor. All source code is valuable down below to this project.

Source code: https://github.com/Vopraan/EEXIST-Vehicle

Permission is hereby granted, free of charge, to any person for this software, hardware, documentation and associated Intellectual Property (the "Design") without limitation to use, copy, modify, merge, publish, distribute, sub-license, build and/or sell. The Design is provided "as is" without warranty of any kind, express or implied.

Bibliography

If you're reading this text online (e.g. in a PDF), all links and references should be clickable; but if you're reading a hardcopy, rather than typing in a long URL, you can go to http://book.songlinesystems.com and click on the reference number (e.g. [2]).

Index

Proof